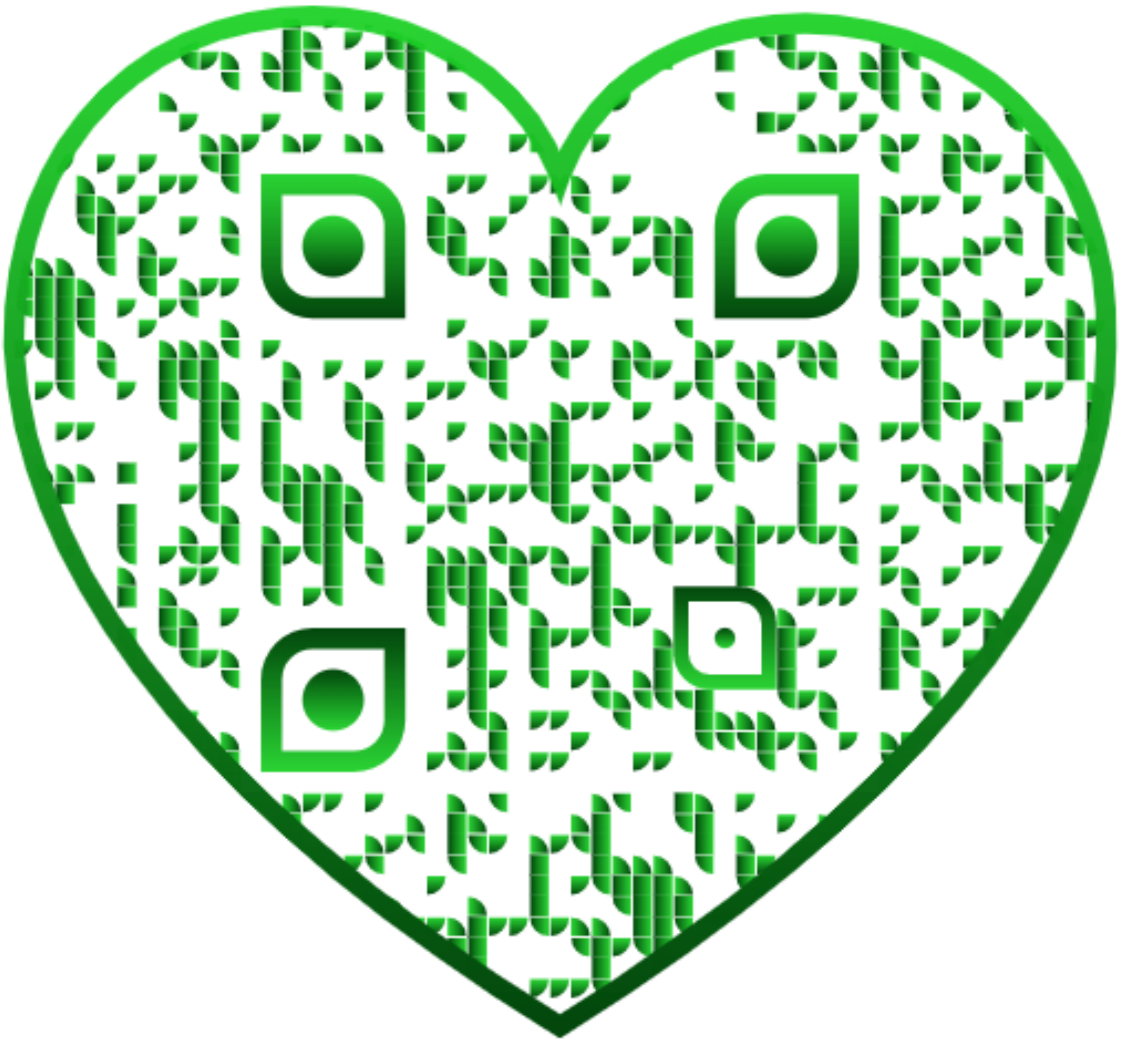
Claude Delannoy

Exercices en LANGAGE C++

178 exercices corrigés

4e édition

EYROLLES



Résumé

178 exercices corrigés pour maîtriser le langage C++

Conçu pour les étudiants en informatique (DUT, licence, master, écoles d'ingénieur), ce recueil d'exercices corrigés et commentés est le complément idéal de *Programmer en langage C++* du même auteur ou de tout autre ouvrage d'initiation au langage C++. L'ouvrage propose 178 exercices pour mieux assimiler la syntaxe de base du langage (types et opérateurs, instructions de contrôle, fonctions, tableaux, pointeurs...) et les concepts objet du C++.

Les exercices vous permettront de vous forger une véritable méthodologie de conception de vos propres classes C++. Vous saurez notamment décider du bien-fondé de la surdéfinition de l'opérateur d'affectation ou du constructeur par recopie, tirer parti de l'héritage (simple ou multiple), créer vos propres bibliothèques de classes, exploiter les possibilités offertes par les patrons de fonctions et de classes, etc.

Cette 4e édition inclut 20 nouveaux exercices portant notamment sur les pointeurs intelligents et sur la nouvelle sémantique de déplacement introduits par les versions C++11 et C++14 de la norme.

Chaque chapitre débute par un rappel de cours suivi de plusieurs exercices de difficulté croissante. Les corrigés sont tous présentés suivant le même canevas : analyse détaillée du problème, solution sous forme de programme avec exemple de résultat d'exécution, justification des choix opérés – car il n'y a jamais de solution unique à un problème donné ! – et, si besoin, commentaires sur les points délicats et suggestions sur les extensions possibles du programme.

Le code source des corrigés est fourni sur le site *www.editions-eyrolles.com.*

Au sommaire

Généralités sur le C++, types de base, opérateurs et expressions (7 exercices) •

Instructions de contrôle (16 exercices) • Fonctions (10 exercices) • Tableaux, pointeurs et chaînes de type C (13 exercices) • Structures (6 exercices) • Du C au C++ (11 exercices) • Classes, constructeurs et destructeurs (7 exercices) • Propriétés des fonctions membres (5 exercices) • Construction, destruction et initialisation des objets (7 exercices) • Fonctions amies (3 exercices) • Surdéfinition d'opérateurs (11 exercices) • Conversions de type définies par l'utilisateur (7 exercices) • Technique de l'héritage (7 exercices) • Héritage multiple (6 exercices) • Fonctions virtuelles (3 exercices) • Flots d'entrée et de sortie (5 exercices) • Patrons de fonctions (4 exercices) • Patrons de classes (8 exercices) • Gestion des exceptions (7 exercices) • Exercices de synthèse (6 exercices) • Composants standard (11 exercices) • Pointeurs intelligents (5 exercices) • Nouvelle sémantique de déplacement (&&) (13 exercices).

Biographie auteur

Ingénieur informaticien au CNRS, **Claude Delannoy** possède une grande pratique de la formation continue et de l'enseignement supérieur. Réputés pour la qualité de leur démarche pédagogique, ses ouvrages sur les langages et la programmation totalisent plus de 500 000 exemplaires vendus.

www.editions-eyrolles.com

AUX ÉDITIONS EYROLLES

Du même auteur

C. DELANNOY. – **Exercices en Java**.
N°67385, 4e édition, 2014, 360 pages (réédition avec nouvelle présentation, 2016).
C. DELANNOY. – **Programmer en langage C++** *(couvre C++11, C++14 et C++17).*
N°67386, 9e édition, 2017, 886 pages.
C. DELANNOY. – **Le guide complet du langage C.**
N°14012, 2014, 844 pages.
C. DELANNOY. – **S'initier à la programmation et à l'orienté objet.**
Avec des exemples en C, C++, C#, Python, Java et PHP.
N°11826, 2e édition, 2014, 360 pages.

Autres ouvrages

H. BERSINI, I. WELLESZ. – **La programmation orientée objet**.
Cours et exercices en UML 2 avec Java, C#, C++, Python, PHP et LinQ.
N°67399, 7e édition, 2017, 696 pages.
P. ROQUES. – **UML 2 par la pratique.**
N°67565, 8e édition, février 2018, 400 pages environ.
G. SWINNEN. – **Apprendre à programmer avec Python 3.**
N°13434, 3e édition, 2012, 435 pages.
J. ENGELS. – **PHP 7.** *Cours et exercices*.
N°67360, 2017, 608 pages.
P. MARTIN, J. PAULI, C. PIERRE de GEYER et E. DASPET. – **PHP 7 avancé.**
N°14357, 2016, 728 pages.
R. RIMELÉ. – **HTML 5.** *Une référence pour le développeur web.*
N°14365, 3e édition, 2017, 820 pages.
H. GIRAUDEL, R. GOETTER. – **CSS 3 : pratique du design web.**
N°14023, 2015, 372 pages.
R. GOETTER. – **CSS 3 Flexbox.**
N°14363, 2016, 152 pages.
E. BIERNAT, M. LUTZ. – **Data science : fondamentaux et études de cas.** *Machine Learning avec Python et R.*
N°14243, 2e édition, mars 2018, 400 pages environ.

Claude Delannoy

Exercices en langage C++

4e édition

EYROLLES

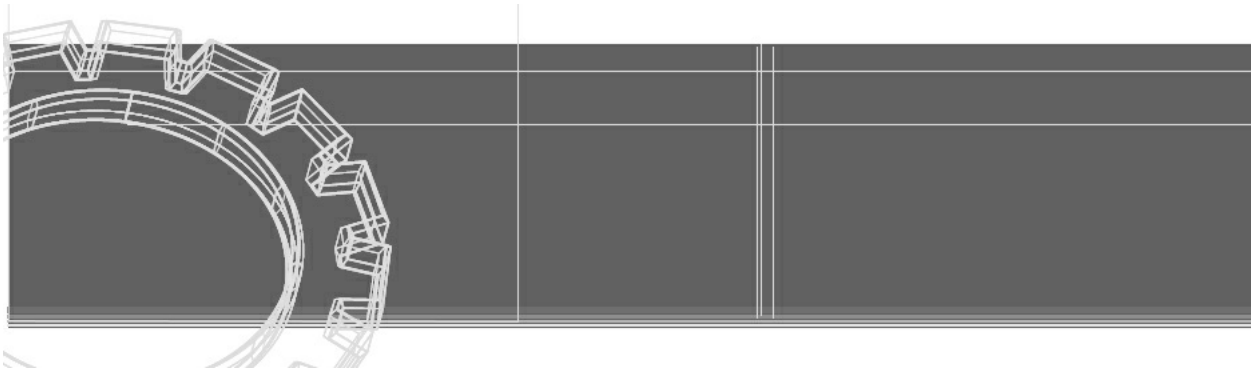
© Groupe EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Attention : pour lire les exemples de lignes de code, réduisez la police de votre support au maximum.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre français d'exploitation du droit de copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 2018.
ISBN : 978-2-212-67663-1

Avant-propos



La maîtrise d'un langage de programmation passe obligatoirement par la pratique, c'est-à-dire la recherche personnelle d'une solution à un problème donné, et cette affirmation reste vraie pour le programmeur chevronné qui étudie un nouveau langage. C'est dans cette situation que se trouve généralement une personne qui aborde le C++ :

- soit elle connaît déjà un langage procédural classique autre que le C (Java, C#, PHP),
- soit elle connaît déjà le langage C sur lequel s'appuie effectivement C++ ; toutefois, ce dernier langage introduit suffisamment de possibilités supplémentaires et surtout de nouveaux concepts (en particulier ceux de la Programmation Orientée Objet) pour que son apprentissage s'apparente à celui d'un nouveau langage.

Cet ouvrage vous propose d'accompagner votre étude du C++ et de la prolonger à l'aide d'exercices appropriés, variés et de difficulté croissante, et ceci quelles que soient vos connaissances préalables. Il comporte :

- 5 chapitres destinés à ceux d'entre vous qui ne connaissent pas le C : types de base, opérateurs et expressions ; instructions de contrôle ; fonctions ; tableaux, pointeurs et chaînes de style C ;
- 1 chapitre pour assurer la transition de C à C++, destiné à ceux qui

connaissent déjà le langage C ;

- 15 chapitres destinés à tous : les notions de classe, constructeur et destructeur ; les propriétés des fonctions membre ; la construction, la destruction et l'initialisation des objets ; les fonctions amies ; la surdéfinition d'opérateurs ; les conversions de type définies par l'utilisateur ; la technique de l'héritage ; les fonctions virtuelles ; les flots d'entrée et de sortie, les patrons de fonctions et les patrons de classes ; la gestion des exceptions.
- 1 chapitre, le 20, qui propose des exercices de synthèse.

Chaque chapitre débute par un rappel détaillé des connaissances nécessaires pour aborder les exercices correspondants (naturellement, un exercice d'un chapitre donné peut faire intervenir des points résumés dans les chapitres précédents).

Le cours complet correspondant à ces résumés se trouve dans l'un de nos ouvrages consacrés au C++, du même auteur, aux éditions Eyrolles.

Au sein de chaque chapitre, les exercices proposés vont d'une application immédiate du cours à des réalisations de classes relativement complètes. Au fil de votre progression dans l'ouvrage, vous réaliserez des classes de plus en plus réalistes et opérationnelles, et ayant un intérêt général.

Citons, par exemple :

- les ensembles ;
- les vecteurs dynamiques ;
- les tableaux dynamiques à plusieurs dimensions ;
- les listes chaînées ;
- les tableaux de bits ;
- les (vraies) chaînes de caractères ;
- les piles ;
- les complexes.

Naturellement, tous les exercices sont corrigés. Pour la plupart, la solution proposée ne se limite pas à une simple liste de programme (laquelle ne représente finalement qu'une rédaction possible parmi d'autres). Vous y

trouverez une analyse détaillée du problème et, si besoin, les justifications de certains choix. Des commentaires viennent, le cas échéant, éclairer les parties quelque peu délicates. Fréquemment, vous trouverez des suggestions de prolongement ou de généralisation du problème abordé.

Outre la maîtrise du langage C++ proprement dit, les exercices proposés vous permettront de vous forger une méthodologie de conception de vos propres classes. Notamment, vous saurez :

- décider du bien-fondé de la surdéfinition de l'opérateur d'affectation ou du constructeur par recopie ;
- exploiter, lorsque vous jugerez que cela est opportun, les possibilités de « conversions implicites » que le compilateur peut mettre en place ;
- tirer parti de l'héritage (simple ou multiple) et déterminer quels avantages présente la création d'une bibliothèque de classes, notamment par le biais du typage dynamique des objets qui découle de l'emploi des fonctions virtuelles ;
- mettre en œuvre les possibilités de fonctions génériques (patrons de fonctions) et de classes génériques (patrons de classes).

Cette nouvelle édition comporte deux chapitres supplémentaires consacrés aux fonctionnalités les plus importantes introduites par C++11 et C++14, à savoir :

- les pointeurs intelligents
- la sémantique dite « de déplacement »

Quelques exercices trouvent une solution évidente en faisant appel aux composants standard introduits par C++98. Nous continuons de les proposer, dans la mesure où la recherche d'une solution ne faisant pas appel aux composants standard conserve un intérêt didactique manifeste. De surcroît, le chapitre 21 montre comment résoudre les exercices lorsqu'on accepte, cette fois, de recourir à ces composants standard.

Table des matières

1 Généralités, types de base, opérateurs et expressions

Exercice 1
Exercice 2
Exercice 3
Exercice 4
Exercice 5
Exercice 6
Exercice 7

2 Les instructions de contrôle

Exercice 8
Exercice 9
Exercice 10
Exercice 11
Exercice 12
Exercice 13
Exercice 14
Exercice 15
Exercice 16
Exercice 17
Exercice 18
Exercice 19
Exercice 20
Exercice 21

Exercice 22
Exercice 23

3 Les fonctions

Exercice 24
Exercice 25
Exercice 26
Exercice 27
Exercice 28
Exercice 29
Exercice 30
Exercice 31
Exercice 32
Exercice 33

4 Les tableaux, les pointeurs et les chaînes de style C

Exercice 34
Exercice 35
Exercice 36
Exercice 37
Exercice 38
Exercice 39
Exercice 40
Exercice 41
Exercice 42
Exercice 43
Exercice 44
Exercice 45
Exercice 46

5 Les structures

Exercice 47
Exercice 48
Exercice 49
Exercice 50
Exercice 51

Exercice 52

6 De C à C++

Exercice 53
Exercice 54
Exercice 55
Exercice 56
Exercice 57
Exercice 58
Exercice 59
Exercice 60
Exercice 61
Exercice 62
Exercice 63

7 Notions de classe, constructeur et destructeur

Exercice 64
Exercice 65
Exercice 66
Exercice 67
Exercice 68
Exercice 69
Exercice 70

8 Propriétés des fonctions membre

Exercice 71
Exercice 72
Exercice 73
Exercice 74
Exercice 75

9 Construction, destruction et initialisation des objets

Exercice 76
Exercice 77
Exercice 78

Exercice 79
Exercice 80
Exercice 81
Exercice 82

10 Les fonctions amies

Exercice 83
Exercice 84
Exercice 85

11 Surdéfinition d'opérateurs

Exercice 86
Exercice 87
Exercice 88
Exercice 89
Exercice 90
Exercice 91
Exercice 92
Exercice 93
Exercice 94
Exercice 95
Exercice 96

12 Les conversions de type définies par l'utilisateur

Exercice 97
Exercice 98
Exercice 99
Exercice 100
Exercice 101
Exercice 102
Exercice 103

13 La technique de l'héritage

Exercice 104
Exercice 105

Exercice 106
Exercice 107
Exercice 108
Exercice 109
Exercice 110

14 L'héritage multiple

Exercice 111
Exercice 112
Exercice 113
Exercice 114
Exercice 115
Exercice 116

15 Les fonctions virtuelles

Exercice 117
Exercice 118
Exercice 119

16 Les flots d'entrée et de sortie

Exercice 120
Exercice 121
Exercice 122
Exercice 123
Exercice 124

17 Les patrons de fonctions

Exercice 125
Exercice 126
Exercice 127
Exercice 128

18 Les patrons de classes

Exercice 129
Exercice 130

Exercice 131
Exercice 132
Exercice 133
Exercice 134
Exercice 135
Exercice 136

19 Gestion des exceptions

Exercice 137
Exercice 138
Exercice 139
Exercice 140
Exercice 141
Exercice 142
Exercice 143

20 Exercices de synthèse

Exercice 144
Exercice 145
Exercice 146
Exercice 147
Exercice 148
Exercice 149

21 Les composants standard

Exercice 150 (69 revisité)
Exercice 151 (70 revisité)
Exercice 152 (79 revisité)
Exercice 153 (80 revisité)
Exercice 154 (81 revisité)
Exercice 155 (92 revisité)
Exercice 156 (93 revisité)
Exercice 157 (95 revisité)
Exercice 158 (144 revisité)
Exercice 159 (145 revisité)
Exercice 160 (96 revisité)

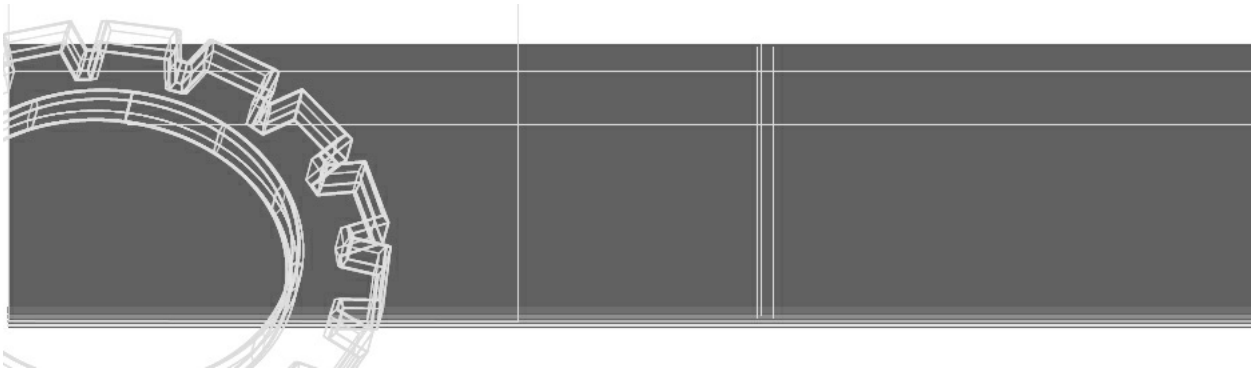
22 Les pointeurs intelligents (C++11/C++14)

Exercice 161
Exercice 162
Exercice 163
Exercice 164
Exercice 165

23 La sémantique de déplacement (C++11/C++14)

Exercice 166
Exercice 167
Exercice 168
Exercice 169
Exercice 170
Exercice 171
Exercice 172
Exercice 173
Exercice 174
Exercice 175
Exercice 176
Exercice 177
Exercice 178

Chapitre 1
Généralités, types de base, opérateurs et expressions



Rappels

Généralités

Le canevas minimal à utiliser pour réaliser un programme en C++ se présente ainsi :

```
#include <iostream>
using namespace std ;
main()          // en-tête
{.....          // corps du programme
}
```

Toute variable utilisée dans un programme doit avoir fait l'objet d'une déclaration en précisant le type et, éventuellement, la valeur initiale. Voici des exemples de déclarations :

```
int i ;  // i est une variable de type int nommée i
```

```
float x = 5.25 ;   // x est une variable de type float nommée
x
                   // initialisée avec la valeur 5.25
const int NFOIS = 5 ;  // NFOIS est une variable de type int
dont la
                       // valeur, fixée à 5, ne peut plus être
modifiée
```

L'affichage d'informations à l'écran est réalisé en envoyant des valeurs sur le « flot *cout* », comme dans :

```
cout << n << 2*p ;  // affiche les valeurs de n et de 2*p sur
l'écran
```

La lecture d'informations au clavier est réalisée en extrayant des valeurs du « flot *cin* », comme dans :

```
cin >> x >> y ;       // lit deux valeurs au clavier et les
affecte à x et à y
```

Types de base

Les types de base sont ceux à partir desquels seront construits tous les autres, dits dérivés (il s'agira des types structurés comme les tableaux, les structures, les unions et les classes, ou d'autres types simples comme les pointeurs ou les énumérations).

Il existe trois types entiers : `short int` (ou `short`), `int` et `long int` (ou `long`). Les limitations correspondantes dépendent de l'implémentation. On peut également définir des types entiers non signés : `unsigned short int` (ou `unsigned short`), `unsigned int` et `unsigned long int` (ou `unsigned long`). Ces derniers sont essentiellement destinés à la manipulation de motifs binaires.

Les constantes entières peuvent être écrites en notation hexadécimale (comme 0xF54B) ou octale (comme 014). On peut ajouter le « suffixe » `u` pour un entier non signé et le suffixe `l` pour un entier de type `long`.

Il existe trois types flottants : `float`, `double` et `long double`. La précision et le « domaine représentable » dépendent de l'implémentation.

Le type « caractère » permet de manipuler des caractères codés sur un octet. Le code utilisé dépend de l'implémentation. Il existe trois types caractère : `signed char`, `unsigned char` et `char` (la norme ne précise pas s'il

correspond à `signed char` ou `unsigned char`).

Les constantes de type caractère, lorsqu'elles correspondent à des « caractères imprimables », se notent en plaçant le caractère correspondant entre apostrophes.

Certains caractères disposent d'une représentation conventionnelle utilisant le caractère « \ », et notamment `'\n'` qui désigne un saut de ligne. De même, `'\''` représente le caractère `'` et `'\"'` désigne le caractère ". On peut également utiliser la notation hexadécimale (comme dans `'\ x41'`) ou octale (comme dans `'\07'`).

Le type `bool` permet de manipuler des « booléens ». Il dispose de deux constantes notées `true` et `false`.

Les opérateurs de C++

Voici un tableau présentant l'ensemble des opérateurs de C++ (certains ne seront exploités que dans les chapitres ultérieurs) :

Catégorie	Opérateurs	Associativité
Résolution de portée	:: (portée globale - unaire) :: (portée de classe - binaire)	<-- -->
Référence	() [] -> .	-->
Unaire	+ - ++ -- ! ~ * & sizeof cast dynamic_cast static_cast reinterpret_cast const_cast new new[] delete delete[]	<---
Sélection	->* .*	<--
Arithmétique	* / %	--->
Arithmétique	+ -	--->
Décalage	<< >>	--->
Relationnels	< <= > >=	--->
Relationnels	== !=	--->
Manipulation de bits	&	--->
Manipulation de	^	--->

bits		
Manipulation de bits	|	--->
Logique	&&	--->
Logique	||	--->
Conditionnel (ternaire)	? :	--->
Affectation	= += -= *= /= %= &= ^= |= <<= >>=	<---
Séquentiel	,	--->

Les opérateurs arithmétiques et les opérateurs relationnels

Les opérateurs arithmétiques binaires (+, -, * et /) et les opérateurs relationnels ne sont définis que pour des opérandes d'un même type parmi `int`, `long int` (et leurs variantes non signées), `float`, `double` et `long double`. Mais on peut constituer des expressions mixtes (opérandes de types différents) ou contenant des opérandes d'autres types (`bool`, `char` et `short`), grâce à l'existence de deux sortes de conversions implicites :

- les conversions d'ajustement de type, selon l'une des hiérarchies :

```
int -> long -> float -> double -> long double
unsinged int -> unsigned long -> float -> double -> long double
```

- les promotions numériques, à savoir des conversions systématiques de `char` (avec ou sans attribut de signe), `bool` et `short` en `int`.

Les opérateurs logiques

Les opérateurs logiques `&&` (et), `||` (ou) et (non) acceptent n'importe quel opérande **numérique** (entier ou flottant) ou pointeur, en considérant que tout opérande de valeur non nulle correspond à « faux » :

Opérande 1	Opérateur	Opérande 2	Résultat
0	&&	0	faux

0	&&	non nul	faux
non nul	&&	0	faux
non nul	&&	non nul	vrai
0	||	0	faux
0	||	non nul	vrai
non nul	||	0	vrai
non nul	||	non nul	vrai
	!	0	vrai
	!	non nul	faux

Les deux opérateurs `&&` et `||` sont « à court-circuit » : le second opérande n'est évalué que si la connaissance de sa valeur est indispensable.

Opérateurs d'affectation

L'opérande de gauche d'un opérateur d'affectation doit être une `lvalue`, c'est-à-dire la référence à quelque chose de modifiable.

Les opérateurs d'affectation (=, -=, += ...), appliqués à des valeurs de type numérique, provoquent la conversion de leur opérande de droite dans le type de leur opérande de gauche. Cette conversion « forcée » peut être « dégradante ».

Opérateurs d'incrémentation et de décrémentation

Les opérateurs unaires d'incrémentation (`++`) et de décrémentation (`--`) agissent sur la valeur de leur unique opérande (qui doit être une `lvalue`) et fournissent la valeur après modification lorsqu'ils sont placés à gauche (comme dans `++n`) ou avant modification lorsqu'ils sont placés à droite (comme dans `n--`).

Opérateur de cast

Il est possible de forcer la conversion d'une expression quelconque dans un type de son choix, grâce à l'opérateur dit de « cast ». Par exemple, si `n` et `p` sont des variables entières, l'expression :

```
(double) n / p        // ou : static_cast<double> (n/p)
```

aura comme valeur celle de l'expression entière `n/p` convertie en `double`.

Opérateur conditionnel

Cet opérateur ternaire fournit comme résultat la valeur de son deuxième opérande si la condition mentionnée en premier opérande est non nulle (vraie pour une expression booléenne), et la valeur de son troisième opérande dans le cas contraire. Par exemple, avec cette affectation :

```
max = a>b ? a : b ;
```

on obtiendra dans la variable `max` la valeur de `a` si la condition `a>b` est vraie, la valeur de `b` dans le cas contraire. Avec :

```
valeur = 2*n-1 ? a : b ;
```

on obtiendra dans la variable `valeur` la valeur de `a` si l'expression `2*n-1` est non nulle, la valeur de `b` dans le cas contraire.

Exercice 1

Énoncé

Éliminer les parenthèses superflues dans les expressions suivantes :

```
a = (x+5)           /* expression 1 */
a = (x=y) + 2       /* expression 2 */
a = (x==y)          /* expression 3 */
(a<b) && (c<d)      /* expression 4 */
(i++) * (n+p)       /* expression 5 */
```

Solution

```
a = x+5             /* expression 1 */
```

L'opérateur `+` est prioritaire sur l'opérateur d'affectation `=`.

```
a = (x=y) + 2       /* expression 2 */
```

Ici, l'opérateur + étant prioritaire sur =, les parenthèses sont indispensables.

```
a = x==y            /* expression 3 */
```

L'opérateur == est prioritaire sur =.

```
a<b && c<d          /* expression 4 */
```

L'opérateur && est prioritaire sur l'opérateur <.

```
i++ * (n+p)         /* expression 5 */
```

L'opérateur ++ est prioritaire sur * ; en revanche, * est prioritaire sur +, de sorte qu'on ne peut éliminer les dernières parenthèses.

Exercice 2

Énoncé

Soient les déclarations :

```
char c = '\x01' ;
short int p = 10 ;
```

Quels sont le type et la valeur de chacune des expressions suivantes :

```
p + 3           /* 1 */
c + 1           /* 2 */
p + c           /* 3 */
3 * p + 5 * c   /* 4 */
```

Solution

1. p est d'abord soumis à la conversion « systématique » `short -> int`, avant d'être ajouté à la valeur 3 (`int`). Le résultat 13 est de type `int`.
2. c est d'abord soumis à la conversion « systématique » `char -> int` (ce qui aboutit à la valeur 1), avant d'être ajouté à la valeur 1 (`int`). Le résultat 2 est de type `int`.
3. p est d'abord soumis à la conversion systématique `short -> int`, tandis que c est soumis à la conversion systématique `char -> int` ; les résultats

sont alors additionnés pour aboutir à la valeur `11` de type `int`.

4. `p` et `c` sont d'abord soumis aux mêmes conversions systématiques que ci-dessus ; le résultat `35` est de type `int`.

Exercice 3

Énoncé

Soient les déclarations :

```
char c = '\x05' ;
int n = 5 ;
long p = 1000 ;
float x = 1.25 ;
double z = 5.5 ;
```

Quels sont le type et la valeur de chacune des expressions suivantes :

```
n + c + p              /* 1 */
2 * x + c              /* 2 */
(char) n + c           /* 3 */
(float) z + n / 2      /* 4 */
```

Solutions

1. `c` est tout d'abord converti en `int`, avant d'être ajouté à `n`. Le résultat (`10`), de type `int`, est alors converti en `long`, avant d'être ajouté à `p`. On obtient finalement la valeur `1010`, de type `long`.

2. On évalue d'abord la valeur de `2*x`, en convertissant `2` (`int`) en `float`, ce qui fournit la valeur `2.5` (de type `float`). Par ailleurs, `c` est converti en `int` (conversion systématique). On évalue ensuite la valeur de `2*x`, en convertissant `2` (`int`) en `float`, ce qui fournit la valeur `2.5` (de type `float`). Pour effectuer l'addition, on convertit alors la valeur entière `5` (`c`) en `float`, avant de l'ajouter au résultat précédent. On obtient finalement la valeur `7.75`, de type `float`.

3. `n` est tout d'abord converti en `char` (à cause de l'opérateur de « cast »),

tandis que `c` est converti (conversion systématique) en `int`. Puis, pour procéder à l'addition, il est nécessaire de reconvertir la valeur de `(char) n` en `int`. Finalement, on obtient la valeur `10`, de type `int`.

4. `z` est d'abord converti en `float`, ce qui fournit la valeur `5.5` (approximative, car, en fait, on obtient une valeur un peu moins précise que ne le serait `5.5` exprimé en `double`). Par ailleurs, on procède à la division entière de `n` par `2`, ce qui fournit la valeur entière `2`. Cette dernière est ensuite convertie en `float`, avant d'être ajoutée à `5.5`, ce qui fournit le résultat `7.5`, de type `float`.

Exercice 4

Énoncé

Soient les déclarations suivantes :

```
int n = 5, p = 9 ;
int q ;
float x ;
```

Quelle est la valeur affectée aux différentes variables concernées par chacune des instructions suivantes ?

```
q = n < p ;                 /* 1 */
q = n == p ;                /* 2 */
q = p % n + p > n ;         /* 3 */
x = p / n ;                 /* 4 */
x = (float) p / n ;         /* 5 */
x = (p + 0.5) / n ;         /* 6 */
x = (int) (p + 0.5) / n ;   /* 7 */
q = n * (p > n ? n : p) ;   /* 8 */
q = n * (p < n ? n : p) ;   /* 9 */
```

Solution

1. `1`
2. `0`

3. `5` (`p%n` vaut `4`, tandis que `p>n` vaut `1`).

4. `1` (`p/n` est d'abord évalué en `int`, ce qui fournit `1` ; puis le résultat est converti en `float`, avant d'être affecté à `x`).

5. `1.8` (`p` est converti en `float`, avant d'être divisé par le résultat de la conversion de `n` en `float`).

6. `1.9` (`p` est converti en `float`, avant d'être ajouté à `0.5` ; le résultat est divisé par le résultat de la conversion de `n` en `float`).

7. `1` (`p` est converti en `float`, avant d'être ajouté à `0.5` ; le résultat (`5.5`) est alors converti en `int` avant d'être divisé par `n`).

8. `25`

9. `45`

Exercice 5

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main ()
{
   int i, j, n ;
   i = 0 ; n = i++ ;
   cout << "A : i = " << i << " n = " << n << "\n" ;

   i = 10 ; n = ++ i ;
   cout << "B : i = " << i << " n = " << n << "\n" ;
   i = 20 ; j = 5 ; n = i++ * ++ j ;
   cout << "C : i = " << i << " j = " << j << " n = " << n
<< "\n" ;
   i = 15 ; n = i += 3 ;
   cout << "D : i = " << i << " n = " << n << "\n" ;

   i = 3 ; j = 5 ; n = i *= --j ;
   cout << "E : i = " << i << " j = " << j << " n = " << n
<< "\n" ;
```

```
}
```

Solution

```
A : i = 1 n = 0
B : i = 11 n = 11
C : i = 21 j = 6 n = 120
D : i = 18 n = 18
E : i = 12 j = 4 n = 12
```

Exercice 6

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{
   int n=10, p=5, q=10, r ;

   r = n == (p = q) ;
   cout << "A : n = " << n << "  p = " << p << " q = " << q
<< " r = " << r << "\n" ;

   n = p = q = 5 ;
   n += p += q ;
   cout << "B : n = " << n << " p = " << p << " q = " << q
<< "\n" ;

   q = n < p ? n++ : p++ ;
   cout << "C : n = " << n << " p = " << p << " q = " << q
<< "\n" ;

   q = n > p ? n++ : p++ ;
   cout << "D : n = " << n << " p = " << p << " q = " << q
<< "\n" ;
```

```
}
```

Solution

```
A : n = 10  p = 10  q = 10  r = 1
B : n = 15  p = 10  q = 5
C : n = 15  p = 11  q = 10
D : n = 16  p = 11  q = 15
```

Exercice 7

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{  int n, p, q ;

   n = 5 ; p = 2 ;                /* cas 1 */
   q = n++ >p || p++ != 3 ;
   cout << "A : n = " << n << " p = " << p << " q = " << q
<< "\n" ;

   n = 5 ; p = 2 ;                /* cas 2 */
   q = n++<p || p++ != 3 ;
   cout << "B : n = " << n << " p = " << p << " q = " << q
<< "\n" ;

   n = 5 ; p = 2 ;                /* cas 3 */
   q = ++n == 3 && ++p == 3 ;
   cout << "C : n = " << n << " p = " << p << " q = " << q
<< "\n" ;

   n = 5 ; p = 2 ;                /* cas 4 */
   q = ++n == 6 && ++p == 3 ;
   cout << "D : n = " << n << " p = " << p << " q = " << q
<< "\n" ;
```

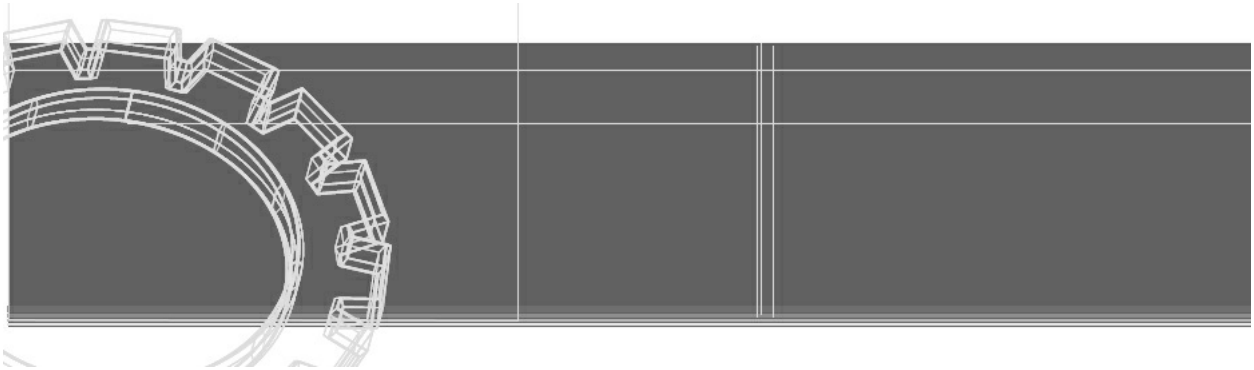
```
}
```

Solution

Il ne faut pas oublier que les opérateurs `&&` et `||` n'évaluent leur second opérande que lorsque cela est nécessaire. Ainsi, ici, il n'est pas évalué dans les cas 1 et 3. Voici les résultats fournis par ce programme :

```
A : n = 6  p = 2  q = 1
B : n = 6  p = 3  q = 1
C : n = 6  p = 2  q = 0
D : n = 6  p = 3  q = 1
```

Chapitre 2
Les instructions de contrôle



Rappels

Le terme *instruction* désignera indifféremment : une instruction simple (terminée par un point-virgule), une instruction structurée (choix, boucle) ou un bloc (instructions entre { et }).

Instruction if

Elle possède deux formes :

```
if (expression) instruction_1
           else instruction_2
if (expression) instruction_1
```

Lorsque des instructions `if` sont imbriquées, un `else` se rapporte toujours au dernier `if` auquel un `else` n'a pas encore été attribué.

Instruction switch

```
switch (expression) { bloc_d_instructions ]
```

Cette instruction évalue la valeur de l'expression entière mentionnée, puis recherche dans le bloc qui suit s'il existe une *étiquette* de la forme `case x` (`x` étant une expression constante, c'est-à-dire calculable par le compilateur) correspondant à cette valeur. Si c'est le cas, il y a branchement à l'instruction figurant à la suite de cette étiquette. Dans le cas contraire, on passe à l'instruction suivant le bloc. L'`expression` peut être de type `char`, auquel cas elle sera convertie en entier.

Une instruction `switch` peut contenir une ou plusieurs instructions `break` qui provoquent la sortie du bloc. Il est possible d'utiliser le mot `default` comme étiquette à laquelle le programme se branche lorsque aucune valeur satisfaisante n'a été rencontrée auparavant.

Instructions do... while et while

```
do instruction while (expression) ;

while (expression) instruction
```

L'expression gouvernant la boucle peut être d'un type quelconque ; elle sera convertie en *bool* selon la règle : non nul devient vrai, nul devient faux.

Instruction for

```
for    ([expression_déclaration_1]    ;    [expression_2]    ;
[expression_3] )
       instruction
```

Les crochets signifient que leur contenu est facultatif.

- `expression_déclaration_1` est soit une expression, soit une déclaration d'une ou plusieurs variables d'un même type, initialisées ou non ;
- `expression_2` est une expression quelconque (qui sera éventuellement convertie en `bool`) ;
- `expression_3` est une expression quelconque.

Cette instruction est équivalente à :

```
{ expression_déclaration_1 ;
  while (expression_2)  { instruction ;
```

```
                        expression_3 ;
                      }
}
```

Note

Depuis C++17, l'expression figurant dans une instruction `if` ou `switch` peut également être une « expression déclaration ».

Instructions break , continue et goto

Une boucle (`do... while`, `while` ou `for`) peut contenir une ou plusieurs instructions `break` dont le rôle est d'interrompre le déroulement de la boucle, en passant à l'instruction qui suit cette boucle. En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne. Si `break` apparaît dans un `switch` imbriqué dans une boucle, elle ne fait sortir que du `switch`.

L'instruction `continue` s'emploie uniquement dans une boucle. Elle permet de passer prématurément au tour de boucle suivant.

L'instruction `goto` permet le branchement en un emplacement quelconque du programme, repéré par une *étiquette*, comme dans cet exemple où, lorsqu'une certaine condition est vraie, on se branche à l'étiquette `erreur` :

```
for (...) { .....
            if (...) goto erreur ;
            .....
          }
   erreur : .....
```

Exercice 8

Énoncé

Quelles erreurs ont été commises dans chacun des groupes d'instructions suivants :

1.

```
if (a<b) cout << "ascendant"
    else cout << "non ascendant" ;
```

2.

```
int n ;
  ...
switch (2*n+1)
{ case 1 : cout << "petit" ;
  case n : cout << "moyen" ;
}
```

3.

```
const int LIMITE=100
int n ;
  ...
switch (n)
{ case LIMITE-1 : cout << "un peu moins" ;
  case LIMITE   : cout << "juste" ;
  case LIMITE+1 : cout << "un peu plus" ;
}
```

Solutions

1. Il manque un point-virgule à la fin de la première ligne :

```
if (a<b) cout << "ascendant" ;
    else cout << "non ascendant" ;
```

2. Les valeurs suivant le mot `case` doivent obligatoirement être des « expressions constantes », c'est-à-dire des expressions calculables par le compilateur lui-même. Ce n'est pas le cas de `n`.

3. Aucune erreur, les expressions telles que `LIMITE-1` étant bien des expressions constantes (ce qui n'était pas le cas en langage C).

Exercice 9

Énoncé

Soit le programme suivant :

```
#include <iostream>
main()
{   int n ;
    cin >> n ;
    switch (n)
    { case 0 : cout << "Nul\n" ;
      case 1 :
      case 2 : cout << "Petit\n" ;
               break ;
      case 3 :
      case 4 :
      case 5 : cout << "Moyen\n" ;
      default : cout << "Grand\n" ;
    }
}
```

Quels résultats affiche-t-il lorsqu'on lui fournit en donnée :

a. 0

b. 1

c. 4

d. 10

e. -5

Solutions

a.

```
Nul
Petit
```

b.

```
Petit
```

c.

```
Moyen
Grand
```

d.

```
Grand
```

e.

```
Grand
```

Exercice 10

Énoncé

Quelles erreurs ont été commises dans chacune des instructions suivantes :

a.

```
do cin >> c while (c != '\n') ;
```

b.

```
do while ( cin >> c, c != '\n') ;
```

c.

```
do {} while (1) ;
```

Solutions

a. Il manque un point-virgule :
```
do cin >> c ; while (c != '\n') ;
```

b. Il manque une instruction (éventuellement « vide ») après le mot `do`. On pourrait écrire, par exemple :
```
do {} while ( (cin >> c, c != '\n') ;
```

ou :
```
do ; while ( cin >> c, c != '\n') ;
```

c. Il n'y aura pas d'erreur de compilation (la valeur entière 1 est convertie en booléen, ce qui fournit la valeur vrai) ; toutefois, il s'agit d'une « boucle infinie ».

Exercice 11

Énoncé

Écrire plus lisiblement :

```
do {} while (cout << "donnez un nombre >0 ", cin >> n,
n<=0) ;
```

Solution

Plusieurs possibilités existent, puisqu'il « suffit » de reporter, dans le corps de la boucle, des instructions figurant « artificiellement » sous forme d'expressions dans la condition de poursuite :

```
do
   cout << donnez un nombre >0 " ;
while (cin >> n, n<=0) ;
```

ou, mieux :

```
do
  { cout << "donnez un nombre >0 " ;
    cin >> n ;
  }
while (n<=0) ;
```

Exercice 12

Énoncé

Soit le petit programme suivant :

```
#include <iostream>
using namespace std ;
main()
{  int i, n, som ;
   som = 0 ;
   for (i=0 ; i<4 ; i++)
      { cout << "donnez un entier " ;
        cin >> n ;
        som += n ;
```

```
        }
     cout << "Somme : " << som ;
  }
```

Écrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction `for` :

a. une instruction `while` ;

b. une instruction `do ... while`.

Solutions

a.

```
#include <iostream>
using namespace std ;
main()
{  int i, n, som ;
   som = 0 ;
    i = 0 ;                          /* ne pas oublier cette
"initialisation" */
   while (i<4)
      { cout << "donnez un entier " ;
        cin >> n ;
        som += n ;
        i++ ;               /* ni cette "incrémentation" */
      }
   cout << "Somme : " << som ;
}
```

b.

```
#include <iostream>
using namespace std ;
main()
{  int i, n, som ;
   som = 0 ;
    i = 0 ;                          /* ne pas oublier cette
"initialisation" */
   do
      { cout << "donnez un entier " ;
        cin >> n ;
        som += n ;
        i++ ;               /* ni cette "incrémentation" */
      }
   while (i<4) ;            /* attention, ici, toujours <4 */
```

```
   cout << "Somme : " << som ;
}
```

Exercice 13

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{  int n=0 ;
   do
     { if (n%2==0) { cout << n << " est pair\n" ;
                     n += 3 ;
                     continue ;
                   }
       if (n%3==0) { cout << n << " est multiple de 3\n" ;
                     n += 5 ;
                   }
       if (n%5==0) { cout << n << " est multiple de 5\n" ;
                     break ;
                   }
       n += 1 ;
     }
   while (1) ;
}
```

Solution

```
0 est pair
3 est multiple de 3
9 est multiple de 3
15 est multiple de 3
20 est multiple de 5
```

Exercice 14

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{   int n, p ;

    n=0 ;
    while (n<=5) n++ ;
    cout << "A : n = " << n << "\n" ;

    n=p=0 ;
    while (n<=8) n += p++ ;
    cout << "B : n = " << n << "\n" ;

    n=p=0 ;
    while (n<=8) n += ++p ;
    cout << "C : n = " << n << "\n" ;

    n=p=0 ;
    while (p<=5) n+= p++ ;
    cout << "D : n = " << n << "\n"  ;

    n=p=0 ;
    while (p<=5) n+= ++p ;
    cout << "E : n = " << n << "\n" ;
}
```

Solution

```
A : n = 6
B : n = 10
C : n = 10
D : n = 15
E : n = 21
```

Exercice 15

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{   int n, p ;

    n=p=0 ;
    while (n<5) n+=2 ; p++ ;
    cout << "A : n = " << n << " p = " << p <<  "\n" ;

    n=p=0 ;
    while (n<5) { n+=2 ; p++ ; }
    cout << "B : n = " << n << " p = " << p << "\n" ;
}
```

Solution

```
A : n = 6, p = 1
B : n = 6, p = 3
```

Exercice 16

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{   int i, n ;

    for (i=0, n=0 ; i<5 ; i++) n++ ;
    cout << "A : i = " << i << " n = " << n << "\n" ;

    for (i=0, n=0 ; i<5 ; i++, n++) {}
```

```
      cout << "B : i = " << i << " n = " << n << "\n" ;

      for (i=0, n=50 ; n>10 ; i++, n-= i ) {}
      cout << "C : i = " << i << " n = " << n << "\n" ;

      for (i=0, n=0 ; i<3 ; i++, n+=i,
         cout << "D : i = " << i << " n = " << n << "\n" ) ;
      cout << "E : i = " << i << " n = " << n << "\n" ;
  }
```

Solution

```
A : i = 5, n = 5
B : i = 5, n = 5
C : i = 9, n = 5
D : i = 1, n = 1
D : i = 2, n = 3
D : i = 3, n = 6
E : i = 3, n = 6
```

Exercice 17

Énoncé

Écrire un programme qui calcule les racines carrées de nombres fournis en donnée. Il s'arrêtera lorsqu'on lui fournira la valeur `0`. Il refusera les valeurs négatives. Son exécution se présentera ainsi :

```
donnez un nombre positif : 2
sa racine carrée est : 1.414214e+00
donnez un nombre positif : -1
svp positif
donnez un nombre positif : 5
sa racine carrée est : 2.236068e+00
donnez un nombre positif : 0
```

Rappelons que la fonction `sqrt` fournit la racine carrée (`double`) de la valeur (`double`) qu'on lui donne en argument.

Solution

Il existe beaucoup de rédactions possibles ; en voici 3 :

1.

```
#include <iostream>
#include <cmath> // pour la déclaration de sqrt
using namespace std ;
main()
{  double x ;
   do
    { cout << "donnez un nombre positif : " ;
      cin >> x ;
      if (x < 0) cout << "svp positif \n" ;
      if (x <=0) continue ;
      cout << "sa racine carrée est : " << sqrt (x) << "\n" ;
    }
   while (x) ;
}
```

2.

```
#include <iostream>
#include <cmath>
using namespace std ;
main()
{  double x ;
   do
     { cout << "donnez un nombre positif : " ;
       cin >> x ;
       if (x<0) { cout << "svp positif \n" ;
                  continue ;
                }
        if (x>0) cout << "sa racine carrée est : " << sqrt (x)
<< "\n" ;
     }
   while (x) ;
}
```

3.

```
#include <iostream>
#include <cmath>
using namespace std ;
main()
{  double x ;
   do
```

```
     { cout <<"donnez un nombre positif : " ;
       cin >> x ;

       if (x < 0) { cout << "svp positif \n" ;
                    continue ;
                  }
        if (x>0) cout << "sa racine carrée est : " << sqrt (x)
<< "\n" ;
       if (x==0) break ;
     }
   while (1) ;
}
```

Exercice 18

Énoncé

Calculer la somme des n premiers termes de la « série harmonique », c'est-à-dire la somme :

1 + 1/2 + 1/3 + 1/4 + ... + 1/n

La valeur de n sera lue en donnée.

Solution

```
#include <iostream>
using namespace std ;
main()
{
    int nt ;              /* nombre de termes de la série
harmonique */
   float som ;       /* pour la somme de la série */
   int i ;

   do
     { cout << "combien de termes : " ;
       cin >> nt ;
     }
   while (nt<1) ;
   for (i=1, som=0 ; i<=nt ; i++) som += (float)1/i ;
```

```
    cout << "Somme des " << nt << " premiers termes = "<< som ;
}
```

Remarques

1. Rappelons que dans :

```
som += (float)1/i
```

l'expression de droite est évaluée en convertissant d'abord `1` et `i` en `float`.

Il faut éviter d'écrire :

```
som += 1/i
```

auquel cas, les valeurs de `1/i` seraient toujours nulles (sauf pour `i=1`) puisque l'opérateur `/`, lorsqu'il porte sur des entiers, correspond à la division entière.

De même, en écrivant :

```
som += (float) (1/i)
```

le résultat ne serait pas plus satisfaisant puisque la conversion en flottant n'aurait lieu qu'après la division (en entier).

En revanche, on pourrait écrire :

```
som += 1.0/i ;
```

2. Si l'on cherchait à exécuter ce programme pour des valeurs élevées de `n` (en prévoyant alors une variable de type `float` ou `double`), on constaterait que la valeur de la somme semble « converger » vers une limite (bien qu'en théorie la série harmonique « diverge »). Cela provient tout simplement de ce que, dès que la valeur de `1/i` est « petite » devant `som`, le résultat de l'addition de `1/i` et de `som` est **exactement** `som`. On pourrait toutefois améliorer le résultat en effectuant la somme « à l'envers » (en effet, dans ce cas, le rapport entre la valeur à ajouter et la somme courante serait plus faible que précédemment).

Exercice 19

Énoncé

Afficher un triangle isocèle formé d'étoiles. La hauteur du triangle (c'est-à-dire le nombre de lignes) sera fourni en donnée, comme dans l'exemple ci-dessous. On s'arrangera pour que la dernière ligne du triangle s'affiche sur le bord gauche de l'écran.

```
combien de lignes ? 10
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
```

Solution

```
#include <iostream>
using namespace std ;

main()
{   const char car = '*' ; /* caractère de remplissage */
    int nlignes ;          /* nombre total de lignes */
    int nl ;               /* compteur de ligne */
     int nesp ;              /* nombre d'espaces précédant une
étoile */
    int j ;

    cout << "combien de lignes ? " ;
    cin >> nlignes ;
    for (nl=0 ; nl<nlignes ; nl++)
       { nesp = nlignes - nl - 1 ;
         for (j=0 ; j<nesp ; j++)   cout << ' ' ;
         for (j=0 ; j<2*nl+1 ; j++) cout << car ;
         cout << '\n' ;
       }
```

```
}
```

Exercice 20

Énoncé

Afficher toutes les manières possibles d'obtenir un euro avec des pièces de 2 cents, 5 cents et 10 cents. Dire combien de possibilités ont été ainsi trouvées. Les résultats seront affichés comme suit :

```
1 euro = 50 X 2c
1 euro = 45 X 2c   2 X 5c
1 euro = 40 X 2c   4 X 5c
1 euro = 35 X 2c   6 X 5c
1 euro = 30 X 2c   8 X 5c
1 euro = 25 X 2c  10 X 5c
1 euro = 20 X 2c  12 X 5c
1 euro = 15 X 2c  14 X 5c
1 euro = 10 X 2c  16 X 5c
1 euro =  5 X 2c  18 X 5c
1 euro = 20 X 5c
1 euro = 45 X 2c   1 X 10c
1 euro = 40 X 2c   2 X 5c   1 X 10c
1 euro = 35 X 2c   4 X 5c   1 X 10c
1 euro = 10 X 2c   2 X 5c   7 X 10c
1 euro =  5 X 2c   4 X 5c   7 X 10c
1 euro =  6 X 5c   7 X 10c
1 euro = 10 X 2c   8 X 10c
1 euro =  5 X 2c   2 X 5c   8 X 10c
1 euro =  4 X 5c   8 X 10c
1 euro =  5 X 2c   9 X 10c
1 euro =  2 X 5c   9 X 10c
1 euro = 10 X 10c

En tout, il y a 66 façons de faire 1 euro.
```

Rappelons que l'insertion dans le flot `cout` d'une expression de la forme `setw(n`, où `n` est une expression entière, demande de réaliser l'affichage suivant (et uniquement ce dernier) sur `n` caractères au minimum. L'emploi de `setw` nécessite l'inclusion du fichier `iomanip`.

Solution

```
#include <iostream>
#include <iomanip>   // pour setw
using namespace std ;
main()
{
    int nbf ;         // compteur du nombre de façons de faire
1 euro
    int n10 ;        // nombre de pièces de 10 centimes
    int n5 ;         // nombre de pièces de 5 centimes
    int n2 ;         // nombre de pièces de 2 centimes
    nbf = 0 ;
    for (n10=0 ; n10<=10 ; n10++)
      for (n5=0 ; n5<=20 ; n5++)
      for (n2=0 ; n2<=50 ; n2++)
      if ( 2*n2 + 5*n5 + 10*n10 == 100)
         { nbf ++ ;
           cout << "1 euro = " ;
           if (n2)  cout << setw(2) << n2  << " X 2c   " ;
           if (n5)  cout << setw(2) << n5  << " X 5c   " ;
           if (n10) cout << setw(2) << n10 << " X 10c  " ;
           cout << "\n" ;
         }

   cout << "\nEn tout, il y a " << nbf << " façons de faire 1
euro\n" ;
}
```

Exercice 21

Énoncé

Écrire un programme qui détermine la $n^{ième}$ valeur u_n (n étant fourni en donnée) de la « suite de Fibonacci » définie comme suit :

$u1 = 1$
$u2 = 1$
$u_n = u_{n-1} + u_{n-2}$ pour n>2

Solution

```
#include <iostream>
using namespace std ;

main()
{
   int u1, u2, u3 ;          /* pour "parcourir" la suite */
   int n ;                   /* rang du terme demandé */
   int i ;                   /* compteur */

   do
     { cout << "rang du terme demandé (au moins 3) ? " ;
       cin >> n ;
     }
   while (n<3) ;

   u2 = u1 = 1 ;             /* les deux premiers termes */
   i = 2 ;
    while (i++ <= n)            /* attention, l'algorithme ne
fonctionne */
      { u3 = u1 + u2 ;        /*            que pour n > 2
        */
        u1 = u2 ;
        u2 = u3 ;
      }
  //             autre formulation possible :
  //  for (i=3 ; i<=n ; i++, u1=u2, u2=u3) u3 = u1 + u2 ;

   cout << "Valeur du terme de rang " << n << " : " << u3 ;
}
```

Notez que, comme à l'accoutumée en C++, beaucoup de formulations sont possibles. Nous en avons d'ailleurs placé une seconde en commentaire de notre programme.

Exercice 22

Énoncé

Écrire un programme qui trouve la plus grande et la plus petite valeur d'une succession de notes (nombres entiers entre `0` et `20`) fournies en données, ainsi que le nombre de fois où ce maximum et ce minimum ont été attribués. On supposera que les notes, en nombre non connu à l'avance, seront terminées par une valeur négative. On s'astreindra à ne pas utiliser de « tableau ». L'exécution du programme pourra se présenter ainsi :

```
donnez une note (-1 pour finir) : 12
donnez une note (-1 pour finir) : 8
donnez une note (-1 pour finir) : 13
donnez une note (-1 pour finir) : 7
donnez une note (-1 pour finir) : 11
donnez une note (-1 pour finir) : 12
donnez une note (-1 pour finir) : 7
donnez une note (-1 pour finir) : 9
donnez une note (-1 pour finir) : -1

note maximale : 13 attribuée 1 fois
note minimale : 7 attribuée 2 fois
```

Solution

```
#include <iostream>
using namespace std ;

main()
{   int note ;      // note "courante"
    int max ;       // note maxi
    int min ;       // note mini
     int nmax ;       // nombre de fois où la note maxi a été
trouvée
     int nmin ;       // nombre de fois où la note mini a été
trouvée
     max = -1 ;    // initialisation max (possible car toutes
notes >=0
     min = 21 ;    // initialisation min (possible car toutes
notes < 21
    while (cout << "donnez une note (-1 pour finir) : ",
           cin >> note, note >=0)
       { if (note == max) nmax++ ;
         if (note > max) { max = note ;
                           nmax = 1 ;
                         }
         if (note == min) nmin++ ;
         if (note < min) { min = note ;
```

```
                        nmin = 1 ;
                      }
      }

   if (max >= 0)
      {  cout << "\nnote maximale : " << max << " attribuée "
              << nmax << " fois\n" ;
         cout << "note minimale : " << min << " attribuée "
              << nmin << " fois\n" ;
      }
    else cout << "vous n'avez fourni aucune note" ;
}
```

Exercice 23

Énoncé

Écrire un programme qui affiche la « table de multiplication » des nombres de `1` à `10`, sous la forme suivante :

```
      I     1     2     3     4     5     6     7     8     9     10
-------------------------------------------------------------------
   1  I     1     2     3     4     5     6     7     8     9     10
   2  I     2     4     6     8    10    12    14    16    18     20
   3  I     3     6     9    12    15    18    21    24    27     30
   4  I     4     8    12    16    20    24    28    32    36     40
   5  I     5    10    15    20    25    30    35    40    45     50
   6  I     6    12    18    24    30    36    42    48    54     60
   7  I     7    14    21    28    35    42    49    56    63     70
   8  I     8    16    24    32    40    48    56    64    72     80
   9  I     9    18    27    36    45    54    63    72    81     90
  10  I    10    20    30    40    50    60    70    80    90    100
```

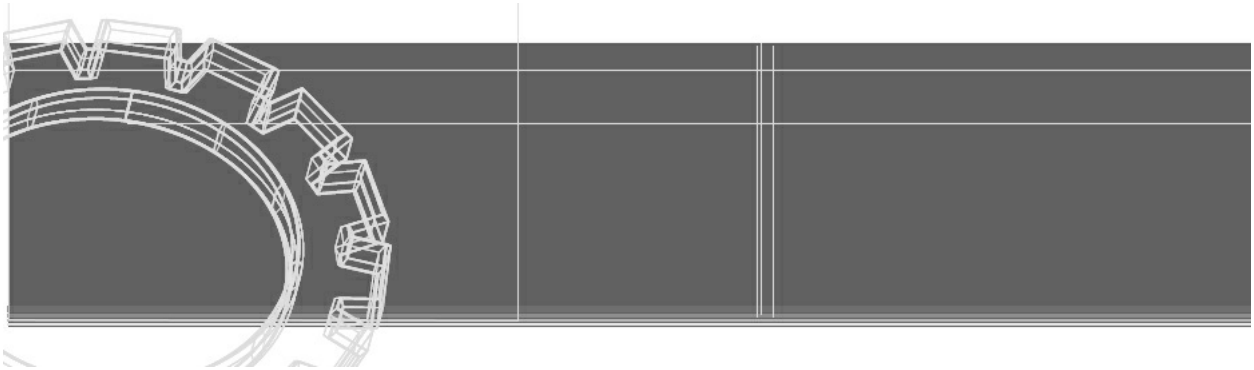
Rappelons que l'insertion dans le flot `cout` d'une expression de la forme `setw(n)`, où `n` est une expression entière, demande de réaliser l'affichage suivant sur `n` caractères au minimum. L'emploi de `setw` nécessite l'inclusion du fichier `iomanip`.

Solution

```
#include <iostream>
#include <iomanip>  // pour setw
```

```
using namespace std ;
main()
{   const int NMAX = 10 ;   // nombre de valeurs
    int i, j ;

         /* affichage ligne en-tête */
    cout << "      I" ;
    for (j=1 ; j<=NMAX ; j++) cout << setw(4) << j  ;
    cout << "\n" ;
    printf ("-------") ;
    for (j=1 ; j<=NMAX ; j++) cout << "----" ;
    cout << "\n" ;

         /* affichage des différentes lignes */
    for (i=1 ; i<=NMAX ; i++)
       { cout << setw(4) << i << "  I" ;
         for (j=1 ; j<=NMAX ; j++)
            cout << setw(4) << i*j ;
         cout << "\n" ;
       }
}
```

Chapitre 3
Les fonctions



Rappels

Généralités

Une fonction est un bloc d'instructions éventuellement paramétré par un ou plusieurs arguments et pouvant fournir un résultat nommé souvent « valeur de retour ». On distingue la définition d'une fonction de son utilisation, cette dernière nécessitant une déclaration.

La définition d'une fonction se présente comme dans cet exemple :

```
float fexple (float x, int b, int c)      // en-tête de la fonction
{  // corps de la fonction
}
```

L'en-tête précise le nom de la fonction (`fexple`) ainsi que le type et le nom (muet) de ses différents arguments (`x`, `b` et `c`). Le corps est un bloc d'instructions qui définit le rôle de la fonction.

Au sein d'une autre fonction (y compris `main`), on utilise cette fonction de cette façon :

```
float fepxle (float, int, int) ;   // déclaration de fexple
("prototype")
   .....
fexple (z, n, p) ; //appel fexple avec les arguments effectifs
z, n et p
```

La déclaration d'une fonction peut être omise lorsqu'elle est connue du compilateur, c'est-à-dire que sa définition a déjà été rencontrée dans le même fichier source.

Mode de transmission des arguments

Par défaut, les arguments sont transmis par valeur. Dans ce cas, les arguments effectifs peuvent se présenter sous la forme d'une expression quelconque.

En faisant suivre du symbole `&` le type d'un argument dans l'en-tête d'une fonction (et dans sa déclaration), on réalise une transmission par référence. Cela signifie que les éventuelles modifications effectuées au sein de la fonction porteront sur l'argument effectif de l'appel et non plus sur une copie. On notera qu'alors l'argument effectif doit obligatoirement être une `lvalue` du même type que l'argument muet correspondant. Toutefois, si l'argument muet est, de surcroît, déclaré avec l'attribut `const`, la fonction reçoit quand même une copie de l'argument effectif correspondant, lequel peut alors être une constante ou une expression d'un type susceptible d'être converti dans le type attendu.

Ces possibilités de transmission par référence s'appliquent également à une valeur de retour, comme nous le verrons au chapitre 6 (dans ce cas, la notion de constance n'a plus de signification).

La notion de référence est théoriquement indépendante de celle de transmission d'argument ; en pratique, elle est rarement utilisée en dehors de ce contexte.

L'instruction return

L'instruction `return` sert à la fois à fournir une valeur de retour et à mettre fin à l'exécution de la fonction. Elle peut mentionner une expression ; elle peut apparaître à plusieurs reprises dans une même fonction ; si aucune instruction `return` n'est mentionnée, le retour est mis en place à la fin de la fonction.

Lorsqu'une fonction ne fournit aucun résultat, son en-tête et sa déclaration comportent le mot `void` à la place du type de la valeur de retour, comme dans :

```
void fSansValRetour (...)
```

Lorsqu'une fonction ne reçoit aucun argument, l'en-tête et la déclaration comportent une liste vide, comme dans :

```
int fSansArguments ()
```

Les arguments par défaut

Dans la déclaration d'une fonction, il est possible de prévoir pour un ou plusieurs arguments (obligatoirement les derniers de la liste) des valeurs par défaut ; elles sont indiquées par le signe `=`, à la suite du type de l'argument, comme dans cet exemple :

```
float fct (char, int = 10, float = 0.0) ;
```

Ces valeurs par défaut seront alors utilisées lorsqu'on appellera ladite fonction avec un nombre d'arguments inférieur à celui prévu. Par exemple, avec la précédente déclaration, l'appel `fct ('a')` sera équivalent à `fct ('a', 10, 0.0)` ; de même, l'appel `fct ('x', 12)` sera équivalent à `fct ('x', 12, 0.0)`. En revanche, l'appel `fct ()` sera illégal.

Conversion des arguments

Lorsqu'un argument est transmis par valeur, il est éventuellement converti dans le type mentionné dans la déclaration (la conversion peut être dégradante). Ces possibilités de conversion disparaissent en cas de transmission par référence : l'argument effectif doit alors être une `lvalue` du type prévu ; ctte dernière ne peut posséder l'attribut `const` si celui-ci n'est

pas prévu dans l'argument muet ; en revanche, si l'agument muet mentionne l'attribut `const`, l'argument effectif pourra être non seulement une constante, mais également une expression d'un type quelconque dont la valeur sera alors convertie dans une variable temporaire dont l'adresse sera fournie à la fonction.

Variables globales et locales

Une variable déclarée en dehors de toute fonction (y compris du `main`) est dite globale. La portée d'une variable globale est limitée à la partie du programme source suivant sa déclaration. Les variable globales ont une classe d'allocation statique, ce qui signifie que leurs emplacements en mémoire restent fixes pendant l'exécution du programme.

Une variable déclarée dans une fonction est dite locale. La portée d'une variable locale est limitée à la fonction dans laquelle elle est définie. Les variables locales ont une classe d'allocation automatique, ce qui signifie que leurs emplacements sont alloués à l'entrée dans la fonction, et libérés à la sortie. Il est possible de déclarer des variables locales à un bloc.

On peut demander qu'une variable locale soit de classe d'allocation statique, en la déclarant à l'aide du mot-clé `static`, comme dans :

```
static int i ;
```

Les variables de classe statique sont, par défaut, initialisées à zéro. On peut les initialiser explicitement à l'aide d'expressions constantes d'un type compatible par affectation avec celui de la variable. Celles de classe automatique ne sont pas initialisées par défaut. Elles doivent être initialisées à l'aide d'une expression quelconque, d'un type compatible par affectation avec celui de la variable.

Surdéfinition de fonctions

En C++, il est possible, au sein d'un même programme, que plusieurs fonctions possèdent le même nom. Dans ce cas, lorsque le compilateur rencontre l'appel d'une telle fonction, il effectue le choix de la « bonne » fonction en tenant compte de la nature des arguments effectifs.

D'une manière générale, si les règles utilisées par le compilateur pour sa

recherche sont assez intuitives, leur énoncé précis est assez complexe, et nous ne le rappellerons pas ici. On trouvera de nombreux exemples de surdéfinition et un récapitulatif complet des règles dans nos ouvrages consacrés au C++ (publiés également aux éditions Eyrolles). Signalons simplement que la recherche d'une fonction surdéfinie peut faire intervenir toutes les conversions usuelles (promotions numériques et conversions standard, ces dernières pouvant être dégradantes), ainsi que les conversions définies par l'utilisateur en cas d'argument de type classe, à condition qu'aucune ambiguïté n'apparaisse.

Les fonctions en ligne

Une fonction en ligne (on dit aussi « développée ») est une fonction dont les instructions sont incorporées par le compilateur (dans le module objet correspondant) à chaaque appel. Cela évite la perte de temps nécessaire à un appel usuel (changement de contexte, copie des valeurs des arguments sur la « pile »...) ; en revanche, les instructions en question sont générées plusieurs fois.

Une fonction en ligne est nécessairement définie en même temps qu'elle est déclarée (elle ne peut plus être compilée séparément) et son en-tête est précédée du mot-clé `inline`, comme dans :

```
inline fct ( ...) {  ..... }
```

Exercice 24

Énoncé

Quelle modification faut-il apporter au programme suivant pour qu'il devienne correct :

```
#include <iostream>
using namespace std ;
main()
{  int n, p=5 ;
   n = fct (p) ;
   cout << "p = " << p << " n = " << n ;
```

```
}
int fct (int r)
{  return 2*r ;
}
```

Solution

La fonction `fct` est utilisée dans la fonction `main`, sans être encore « connue » du compilateur, ce qui provoque une erreur de compilation. Pour y remédier, on dispose de deux possibilités :

- déclarer la fonction avant son utilisation dans `main`, de préférence au début :

```
#include <iostream>
using namespace std ;
main()
{  int fct (int) ;   // on pourrait écrire int fct (int x)
   int n, p=5 ;
   n = fct (p) ;
   cout << "p = " << p << " n = " << n ;
}
int fct (int r)
{  return 2*r ;
}
```

- placer la définition de la fonction avant celle de la fonction `main` :

```
#include <iostream>
using namespace std ;
int fct (int r)
{  return 2*r ;
}
main()
{  int n, p=5 ;
   n = fct (p) ;
   cout << "p = " << p << " n = " << n ;
}
```

Il est conseillé d'utiliser la première démarche qui a le mérite de permettre de modifier les emplacements des définitions de fonctions dans un fichier source ou, même, de le scinder en plusieurs parties (compilation séparée).

Exercice 25

Énoncé

Écrire :

- une fonction, nommée `f1`, se contentant d'afficher « `bonjour` » (elle ne possédera aucun argument, ni valeur de retour) ;
- une fonction, nommée `f2`, qui affiche « `bonjour` » un nombre de fois égal à la valeur reçue en argument (`int`) et qui ne renvoie aucune valeur ;
- une fonction, nommée `f3`, qui fait la même chose que `f2`, mais qui, de plus, renvoie la valeur (`int`) `0`.

Écrire un petit programme appelant successivement chacune de ces 3 fonctions, après les avoir convenablement déclarées (on ne fera aucune hypothèse sur les emplacements relatifs des différentes fonctions composant le fichier source).

Solution

L'énoncé ne précisant rien, nous utiliserons une transmission d'arguments par valeur. Comme on n'impose pas d'ordre aux définitions des différentes fonctions dans le fichier source, on déclarera systématiquement toutes les fonctions utilisées.

```
#include <iostream>
using namespace std ;

void f1 (void)
{  cout << "bonjour\n" ;
}

void f2 (int n)
{  int i ;
   for (i=0 ; i<n ; i++)
       cout << "bonjour\n" ;
}
```

```
int f3 (int n)
{  int i ;
   for (i=0 ; i<n ; i++)
       cout << "bonjour\n" ;
   return 0 ;
}

main()
{  void f1 (void) ;
   void f2 (int) ;
   int f3 (int) ;
   f1 () ;
   f2 (3) ;
   f3 (3) ;
}
```

Exercice 26

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;

int n=10, q=2 ;
main()
{
   int fct (int) ;
   void f (void) ;
   int n=0, p=5 ;
   n = fct(p) ;
   cout << "A : dans main, n = " << n << " p = " << p
        << " q = " << q << "\n" ;
   f() ;
}

int fct (int p)
{
   int q ;
   q = 2 * p + n ;
   cout << "B : dans fct,  n = " << n << " p = " << p
```

```
          << " q = " << q << "\n" ;
    return q ;
}

void f (void)
{
    int p = q * n ;
    cout << "C : dans f,    n = " << n << " p = " << p
         << " q = " << q << "\n" ;
}
```

Solution

```
B : dans fct,  n = 10, p = 5, q = 20
A : dans main, n = 20, p = 5, q = 2
C : dans f,    n = 10, p = 20, q = 2
```

Exercice 27

Énoncé

Écrire une fonction qui reçoit en arguments 2 nombres flottants et un caractère, et qui fournit un résultat correspondant à l'une des 4 opérations appliquées à ses deux premiers arguments, en fonction de la valeur du dernier, à savoir : addition pour le caractère `+`, soustraction pour `-`, multiplication pour `*` et division pour `/` (tout autre caractère que l'un des 4 cités sera interprété comme une addition). On ne tiendra pas compte des risques de division par zéro.

Écrire un petit programme (`main`) utilisant cette fonction pour effectuer les 4 opérations sur les *2* nombres fournis en donnée.

Solution

```
#include <iostream>
using namespace std ;
```

```
float oper (float v1, float v2, char op)
{   float res ;
    switch (op)
    { case '+' : res = v1 + v2 ;
                 break ;
      case '-' : res = v1 - v2 ;
                 break ;
      case '*' : res = v1 * v2 ;
                 break ;
      case '/' : res = v1 / v2 ;
                 break ;
      default  : res = v1 + v2 ;
    }
    return res ;
}

main()
{   float oper (float, float, char) ;       // déclaration de
oper
    float x, y ;

    cout << "donnez deux nombres réels : " ;
    cin >> x >> y ;

     cout << "leur somme est :       " << oper (x, y, '+') <<
"\n" ;
     cout << "leur différence est : " << oper (x, y, '-') <<
"\n" ;
     cout << "leur produit est :     " << oper (x, y, '*') <<
"\n" ;
     cout << "leur quotient est :    " << oper (x, y, '/') <<
"\n" ;
}
```

Exercice 28

Énoncé

Transformer le programme (fonction + `main`) écrit dans l'exercice précédent de manière que la fonction ne dispose plus que de 2 arguments, le caractère indiquant la nature de l'opération à effectuer étant précisé,

cette fois, à l'aide d'une variable globale.

Solution

```
#include <iostream>
using namespace std ;
char  op  ;          //  variable  globale  pour  la  nature  de
l'opération
                // attention : doit être déclarée avant d'être
utilisée

float oper (float v1, float v2)
{   float res ;
    switch (op)
    { case '+' : res = v1 + v2 ;
                 break ;
      case '-' : res = v1 - v2 ;
                 break ;
      case '*' : res = v1 * v2 ;
                 break ;
      case '/' : res = v1 / v2 ;
                 break ;
      default  : res = v1 + v2 ;
    }
    return res ;
}

main()
{   float oper (float, float) ;      /* prototype de oper */
    float x, y ;

    cout << "donnez deux nombres réels : " ;
    cin >> x >> y ;

    op = '+' ;
    cout << "leur somme est :      " << oper (x, y) << "\n" ;
    op = '-' ;
    cout << "leur différence est : " << oper (x, y) << "\n" ;
    op = '*' ;
    cout << "leur produit est :    " << oper (x, y) << "\n" ;
    op = '/' ;
    cout << "leur quotient est :   " << oper (x, y) << "\n" ;
}
```

Remarque

Il s'agissait ici d'un exercice d'« école » destiné à forcer l'utilisation d'une variable globale. Dans la pratique, on évitera autant que possible ce genre de programmation qui favorise trop les risques d'« effets de bord ».

Exercice 29

Énoncé

Écrire une fonction, sans argument ni valeur de retour, qui se contente d'afficher, à chaque appel, le nombre total de fois où elle a été appelée sous la forme :

```
appel numéro 3
```

Solution

La meilleure solution consiste à prévoir, au sein de la fonction en question, une variable de classe statique. Elle sera initialisée une seule fois à zéro (ou à toute autre valeur éventuellement explicitée) au début de l'exécution du programme. Ici, nous avons, de plus, prévu un petit programme d'essai.

```
#include <iostream>
using namespace std ;

void fcompte (void)
{
    static int i ;      // il est inutile, mais pas interdit,
d'écrire i=0
   i++ ;
   cout << "appel numéro " << i << "\n"  ;
}

     /* petit programme d'essai de fcompte */
main()
```

```
{  void fcompte (void) ;
   int i ;
   for (i=0 ; i<3 ; i++) fcompte () ;
}
```

Là encore, la démarche consistant à utiliser comme compteur d'appels une variable globale (qui devrait alors être connue du programme utilisateur) est à proscrire.

Exercice 30

Énoncé

Écrire 2 fonctions à un argument entier et une valeur de retour entière permettant de préciser si l'argument reçu est multiple de `2` (pour la première fonction) ou multiple de `3` (pour la seconde fonction).

Utiliser ces deux fonctions dans un petit programme qui lit un nombre entier et qui précise s'il est pair, multiple de `3` et/ou multiple de `6`, comme dans cet exemple (il y a deux exécutions) :

```
donnez un entier : 9
il est multiple de 3

   ---------------
donnez un entier : 12
il est pair
il est multiple de 3
il est divisible par 6
```

Solution

```
#include <iostream>
using namespace std ;

int mul2 (int n)
{  if (n%2) return 0 ;
       else return 1 ;
}
```

```
int mul3 (int n)
{  if (n%3) return 0 ;
       else return 1 ;
}

main()
{  int mul2 (int) ;
   int mul3 (int) ;
   int n ;
   cout << "donnez un entier : " ;
   cin >> n ;
   if (mul2(n))            cout << "il est pair\n" ;
   if (mul3(n))            cout << "il est multiple de 3\n" ;
    if (mul2(n) && mul3(n)) cout << "il est divisible par
6\n" ;
}
```

Exercice 31

Énoncé

Écrire une fonction permettant d'ajouter une valeur fournie en argument à une variable fournie également en argument. Par exemple, l'appel (`n` et `p` étant entiers) :

```
ajouter (2*p+1, n) ;
```

ajoutera la valeur de l'expression `2*p+1` à la variable `n`.

Écrire un petit programme de test de la fonction.

Solution

Étant donné que la fonction doit être en mesure de modifier la valeur de son second argument, il est nécessaire que ce dernier soit transmis par référence.

```
#include <iostream>
using namespace std ;

void ajoute (int exp, int & var)
{ var += exp ;
```

```
  return ;
}
main()
{ void ajoute (int, int &) ;
  int n = 12 ;
  int p = 3 ;
  cout << "Avant, n = " << n << "\n" ;
  ajoute (2*p+1, n) ;
  cout << "Après, n = " << n << "\n" ;
}
```

Exercice 32

Énoncé

Soient les déclarations suivantes :

```
int fct (int) ;          // fonction I
int fct (float) ;        // fonction II
void fct (int, float) ;  // fonction III
void fct (float, int) ;  // fonction IV
int n, p ;
float x, y ;
char c ;
double z ;
```

Les appels suivants sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions éventuellement mises en place ?

a. `fct (n) ;`

b. `fct (x) ;`

c. `fct (n, x) ;`

d. `fct (x, n) ;`

e. `fct (c) ;`

f. `fct (n, p) ;`

g. `fct (n, c) ;`

h. `fct (n, z) ;`

i. `fct (z, z) ;`

Solutions

Les cas **a**, **b**, **c** et **d** ne posent aucun problème. Il y a respectivement appel des fonctions I, II, III et IV, sans qu'aucune conversion d'argument ne soit nécessaire.

e. Appel de la fonction I, après conversion de la valeur de `c` en `int`.

f. Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : conserver `n`, convertir `p` en `float` et appeler la fonction III ou, au contraire, convertir `n en float`, conserver `p` et appeler la fonction IV.

g. Appel de la fonction III, après conversion de `c` en `float`.

h. Appel de la fonction III, après conversion (dégradante) de `z` en `float`.

i. Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : convertir le premier argument en `float` et le second en `int` et appeler la fonction III ou, au contraire, convertir le premier argument `en int` et le second `en float` et appeler la fonction IV.

Exercice 33

Énoncé

a. Transformer le programme suivant pour que la fonction `fct` devienne une fonction en ligne.

```
#include <iostream>
using namespace std ;
main()
{   int  fct  (char,  int)  ;                  // déclaration
(prototype) de fct
   int n = 150, p ;
   char c = 's' ;
   p = fct ( c , n) ;
    cout << "fct (\'" << c << "\', " << n << ") vaut : " <<
p ;
```

```
}
int fct (char c, int n)            // définition de fct
{  int res ;
   if (c == 'a')      res = n + c ;
   else if (c == 's') res = n - c ;
   else               res = n * c ;
   return res ;
}
```

b. Comment faudrait-il procéder si l'on souhaitait que la fonction `fct` soit compilée séparément ?

Solutions

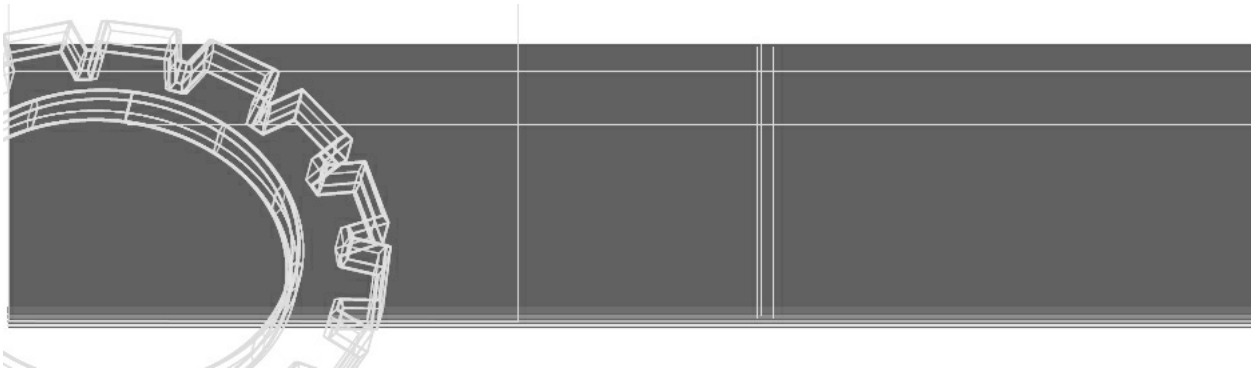
a. Nous devons donc d'abord déclarer (et définir en même temps) la fonction `fct` comme une fonction en ligne. Le programme `main` s'écrit de la même manière, si ce n'est que la déclaration de `fct` n'y est plus nécessaire puisqu'elle apparaît auparavant (il reste permis de la déclarer, à condition de ne pas utiliser le qualificatif `inline`).

```
#include <iostream>
using namespace std ;
inline int fct (char c, int n)
{   int res ;
    if (c == 'a')      res = n + c ;
    else if (c == 's') res = n - c ;
    else               res = n * c ;
    return res ;
}
main ()
{   int n = 150, p ;
    char c = 's' ;
    p = fct (c, n) ;
     cout << "fct (\'" << c << "\', " << n << ") vaut : " <<
p ;
}
```

b. Il s'agit en fait d'une question piège. En effet, la fonction `fct` étant en ligne, elle ne peut plus être compilée séparément. Il est cependant possible de la conserver dans un fichier d'extension `h` et d'incorporer simplement ce fichier par `#include` pour compiler le `main`. Cette démarche se rencontrera d'ailleurs fréquemment dans le cas de classes comportant des fonctions en ligne. Alors, dans un fichier d'extension `h`, on trouvera la

déclaration de la classe en question, à l'intérieur de laquelle apparaîtront les « déclarations-définitions » des fonctions en ligne.

Chapitre 4
Les tableaux, les pointeurs et les chaînes de style C



Rappels

Tableau à un indice

Un tableau à un indice est un ensemble d'éléments de même type désignés par un identificateur unique. Chaque élément est repéré par un indice précisant sa position dans l'ensemble (le premier élément est repéré par l'indice 0).

L'instruction suivante :

```
float t [10] ;
```

réserve l'emplacement pour un tableau de 10 éléments de type `float`, nommé `t`.

Un élément de tableau est une `lvalue`. Un indice peut prendre la forme de n'importe quelle expression arithmétique d'un type entier quelconque. Le

nombre d'éléments d'un tableau (dimension) doit être indiqué sous forme d'une expression constante.

Tableaux à plusieurs indices

La déclaration :

```
int t[5][3] ;
```

réserve un tableau de 15 (5 × 3) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :

```
t[3][2]     t[i][j]
```

D'une manière générale, on peut définir des tableaux comportant un nombre quelconque d'indices.

Les éléments d'un tableau sont rangés en mémoire selon l'ordre obtenu en faisant varier le dernier indice en premier.

Initialisation des tableaux

Les tableaux de classe statique sont, par défaut, initialisés à zéro. Les tableaux de classe automatique ne sont pas initialisés par défaut.

On peut initialiser (partiellement ou totalement) un tableau lors de sa déclaration, comme dans ces exemples (le signe = peut être supprimé depuis C++11) :

```
int t1[5] = { 10, 20, 5, 0, 3 } ;
       // place les valeurs 10, 20, 5, 0 et 3 dans les cinq
éléments de t1

int t2 [5] = { 10, 20 } ;
      // place les valeurs 10 et 20 dans les deux premiers
éléments de t2
```

Les deux déclarations suivantes sont équivalentes :

```
int tab [3] [4] = { { 1, 2, 3, 4 },
                    { 5, 6, 7, 8 },
                    { 9,10,11,12 }  }
int tab [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;
```

Les initialiseurs fournis pour l'initialisation des éléments d'un tableau doivent être des expressions constantes pour les tableaux de classe statique, et

des expressions quelconques (d'un type compatible par affectation) pour les tableaux de classe automatique.

Les pointeurs, les opérateurs & et *

Une variable pointeur est destinée à manipuler des adresses d'informations d'un type donné. On la déclare comme dans ces exemples :

```
int * adi ;     // adi contiendra des adresses d'entiers de
type int
float * adf ;  // adf contiendra des adresses de flottants de
type float
```

L'opérateur `&` permet d'obtenir l'adresse d'une variable :

```
int n ;
   .....
adi = &n ;          // adi contient l'adresse de n
```

L'opérateur `*` permet d'obtenir l'information d'adresse donnée. Ainsi `*adi` désigne la `lvalue` d'adresse `adi`, c'est-à-dire ici `n` :

```
int p = *adi ;      // place dans p la valeur de n
*adi = 40 ;          // place la valeur 40 à l'adresse contenue
dans adi,
                    // donc ici dans n
```

Il existe un type « pointeur générique », c'est-à-dire pour lequel on ne précise pas la nature des informations pointées, à savoir le type `void *`.

Opérations sur les pointeurs

On peut incrémenter ou décrémenter des pointeurs d'une valeur donnée. Par exemple :

```
adi++ ;             // modifie adi de manière qu'elle pointe
                    // sur l'entier suivant n
adi -= 10 ;         // modifie adi de manière qu'elle pointe
                    // 10 entiers avant
```

L'unité d'incrémentation ou de décrémentation des pointeurs génériques est l'octet.

On peut comparer ou soustraire des pointeurs de même type (pointant sur des éléments de même type).

Affectations de pointeurs et conversions

Il n'existe aucune conversion implicite d'un type pointeur en un autre, à l'exception de la conversion en pointeur générique.

Il n'existe aucune conversion implicite d'un entier en un pointeur, à l'exception de la valeur 0 (ou `nullptr` depuis C++11) qui est alors convertie en un pointeur ne « pointant sur rien ».

Tableaux et pointeurs

Un nom de tableau est une constante pointeur. Avec :

```
int t[10] ;
```

`t+1` est équivalent à `&t[1]` ;

`t+i` est équivalent à `&t[i]` ;

`ti]` est équivalent à `*(t+i)`.

Avec :

```
int t[3] [4] ;
```

`t` est équivalent à `&t[0][0]` ou à `t[0]` ;

`t+2` est équivalent à `&t[2][0]` ou à `t[2]`

Lorsqu'un nom de tableau est utilisé en argument effectif, c'est (la valeur de) l'adresse qui est transmise à la fonction. La notion de transmission par référence n'a pas de sens dans ce cas. Dans la déclaration d'une fonction disposant d'un argument de type tableau, on peut utiliser indifféremment le formalisme « tableau » (avec ou sans la dimension effective du tableau) ou le formalisme pointeur ; ces trois déclarations sont équivalentes :

```
void fct (int t[10])
void fct (int t[])
void fct (int * t)
```

Gestion dynamique de la mémoire

Si `type` représente la description d'un type absolument quelconque, on peut allouer un emplacement pour un élément de type `type` par :

```
new type
```

On peut également allouer l'emplacement nécessaire pour ***n* éléments** du type `type` par :

```
new type [n]
```

Le résultat est un pointeur (de type `type *`) sur l'élément correspondant (ou le premier élément dans le deuxième cas). En cas d'échec d'allocation, la norme laisse l'implémentation libre de choisir entre le renvoi d'un pointeur de valeur nulle ou le déclenchement d'une exception `bad_alloc` (les exceptions font l'objet du chapitre 15).

Si `adr` désigne l'adresse de l'emplacement ainsi alloué, sa libération se fait à l'aide de l'une des deux syntaxes :

```
delete adr        // pour les éléments simples
delete [] adr     // pour les tableaux d'éléments
```

Pointeurs sur des fonctions

Un pointeur sur une fonction est défini par le type des arguments et de la valeur de retour de la fonction, comme dans :

```
int (* adf) (double, int) ;   // adf est un pointeur sur une
fonction
                              // recevant un double et un int
                              // et renvoyant un int
```

Un nom de fonction, employé seul, est traduit par le compilateur en l'adresse de cette fonction.

Chaînes de style C

Une chaîne de caractère de style C est une suite d'octets (représentant chacun un caractère), terminée par un octet de code nul. Les chaînes constantes sont représentées selon cette convention. Les lectures opérées sur le flot `cin`, ainsi que les écritures sur le flot `cout`, utilisent cette convention lorsqu'elles portent sur des `lvalue` de type tableau de caractères ou pointeur sur des caractères (type `char *`).

Un tableau de caractères peut être initialisé par une chaîne constante. Ces deux instructions sont équivalentes :

```
char ch[20] = "bonjour" ;
char ch[20] = { 'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0') ;
```

Arguments de la ligne de commande

Lors du lancement d'un programme, la plupart des environnements permettent de lui fournir des « paramètres ». Ceux-ci sont alors simplement transmis à la fonction `main`, sous forme de chaîne de style C, comme des arguments effectifs d'appel d'une fonction. Ils sont, en outre, précédés du nombre total d'arguments (`int`) et du nom du programme (chaîne de style C).

Pour pouvoir les exploiter, il est alors nécessaire d'utiliser une déclaration de `main` telle que :

```
main (int nbarg, char * argv[])
```

Exercice 34

Énoncé

Quels résultats fournira ce programme :

```
#include <stdio.h>

#include <iostream>
using namespace std ;

main()
{
   int t [3] ;
   int i, j ;
   int * adt ;

    for (i=0, j=0 ; i<3 ; i++) t[i] = j++ + i ;
       /* 1 */

    for (i=0 ; i<3 ; i++) cout << t[i] << " " ;
       /* 2 */
   cout << "\n" ;

    for (i=0 ; i<3 ; i++) cout << *(t+i) << " " ;
      /* 3 */
   printf ("\n") ;
```

```
    for (adt = t ; adt < t+3 ; adt++) cout << *adt << "
"  ;   /* 4 */
   cout << "\n" ;

    for (adt = t+2 ; adt>=t ; adt--) cout << *adt << "
";      /* 5 */
   cout << "\n" ;
}
```

Solution

`/* 1 */` remplit le tableau avec les valeurs `0 (0+0)`, `2 (1+1)` et `4 (2+2)` ; on obtiendrait plus simplement le même résultat avec l'expression `2*i`.

`/* 2 */` affiche classiquement les valeurs du tableau t, dans l'ordre naturel.

`/*  3  */` fait la même chose, en utilisant le formalisme pointeur au lieu du formalisme tableau. Ainsi, `*(t+i)` est parfaitement équivalent à `t[i]`.

`/* 4 */` fait la même chose, en utilisant la `lvalue adt` (à laquelle on a affecté initialement l'adresse `t` du tableau) et en l'incrémentant pour parcourir les différentes adresses des 4 éléments du tableau.

`/* 5 */` affiche les valeurs de `t`, à l'envers, en utilisant le même formalisme pointeur que dans 4. On aurait pu écrire, de façon équivalente :

```
for (i=2 ; i>=0 ; i--) cout << t[i] << " " ;
```

Voici les résultats fournis par ce programme :

```
0 2 4
0 2 4
0 2 4
4 2 0
```

Exercice 35

Énoncé

Écrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit :

a. en utilisant uniquement le « formalisme tableau » ;

b. en utilisant le « formalisme pointeur », à chaque fois que cela est possible.

Solutions

a. La programmation est, ici, classique. Nous avons simplement défini une constante `NVAL` destinée à contenir le nombre de valeurs du tableau. Notez bien que la déclaration `int t[NVAL]` est acceptée puisque `NVAL` est une « expression constante ».

```
#include <iostream>
using namespace std ;

main()
{   const int NVAL = 10 ;                /* nombre de valeurs
du tableau */
    int i, min, max ;
    int t[NVAL] ;

    cout << "donnez " << NVAL << " valeurs\n"  ;
    for (i=0 ; i<NVAL ; i++) cin >> t[i] ;

    max = min = t[0] ;
    for (i=1 ; i<NVAL ; i++)
       { if (t[i] > max) max = t[i] ;  /* ou max = t[i]>max ?
t[i] : max */
         if (t[i] < min) min = t[i] ;  /* ou min = t[i]<min ?
t[i] : min */
      }

    cout << "valeur max : " << max << "\n" ;
    cout << "valeur min : " << min << "\n" ;
}
```

b. On peut remplacer systématiquement `t[i]` par `*(t+i)`. Voici finalement le programme obtenu :

```
#include <iostream>
using namespace std ;

main()
```

```
{   const int NVAL = 10 ;                /* nombre de valeurs
du tableau */
    int i, min, max ;
    int t[NVAL] ;

    cout << "donnez " << NVAL << " valeurs\n"  ;
    for (i=0 ; i<NVAL ; i++) cin >> *(t+i) ;

    max = min = *t ;
    for (i=1 ; i<NVAL ; i++)
      { if ( *(t+i) > max) max = *(t+i) ;
        if ( *(t+i) < min) min = *(t+i) ;
      }

    cout << "valeur max : " << max << "\n" ;
    cout << "valeur min : " << min << "\n" ;
}
```

Exercice 36

Énoncé

Soient deux tableaux `t1` et `t2` déclarés ainsi :

```
float t1[10], t2[10] ;
```

Écrire les instructions permettant de recopier, dans `t1`, tous les éléments positifs de `t2`, en complétant éventuellement `t1` par des zéros. Ici, on ne cherchera pas à fournir un programme complet et on utilisera systématiquement le formalisme tableau.

Solution

On peut commencer par remplir `t1` de zéros, avant d'y recopier les éléments positifs de `t2` :

```
int i, j ;
for (i=0 ; i<10 ; i++) t1[i] = 0 ;
   /* i sert à pointer dans t1 et j dans t2 */
```

```
for (i=0, j=0 ; j<10 ; j++)
    if (t2[j] > 0) t1[i++] = t2[j] ;
```

Mais on peut recopier d'abord dans `t1` les éléments positifs de `t2`, avant de compléter éventuellement par des zéros. Cette seconde formulation, moins simple que la précédente, se révélerait toutefois plus efficace sur de grands tableaux :

```
int i, j ;
for (i=0, j=0 ; j<10 ; j++)
    if (t2[j] > 0) t1[i++] = t2[j] ;
for (j=i ; j<10 ; j++) t1[j] = 0 ;
```

Exercice 37

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;

main()
{  int t[4] = {10, 20, 30, 40} ;
   int * ad [4] ;
   int i ;
    for (i=0 ; i<4 ; i++) ad[i] = t+i ;
   /* 1 */
    for (i=0 ; i<4 ; i++) cout << * ad[i] << " " ;
 /* 2 */
   cout << "\n" ;
     cout <<   * (ad[1] + 1) << " " << * ad[1] + 1 <<
"\n" ;  /* 3 */
}
```

Solution

Le tableau `ad` est un tableau de 4 éléments ; chacun de ces éléments est un pointeur sur un `int`. L'instruction `/* 1 */` remplit le tableau `ad` avec les adresses des 4 éléments du tableau `t`. L'instruction `/* 2 */` affiche

finalement les 4 éléments du tableau `t` ; en effet, `* ad[i]` représente la valeur située à l'adresse `ad[i]`. `/* 2 */` est équivalente ici à :

```
for (i=0 ; i<4 ; i++) cout << t[i] << " " ;
```

Enfin, dans l'instruction `/* 3 */`, `*(ad[1] + 1)` représente la valeur située à l'entier suivant celui d'adresse `ad[1]` ; il s'agit donc de `t[2]`. En revanche, `*ad[1] + 1` représente la valeur située à l'adresse `ad[1]` augmentée de `1`, autrement dit `t[1] + 1`.

Voici, en définitive, les résultats fournis par ce programme :

```
10 20 30 40
30 21
```

Exercice 38

Énoncé

Soit le tableau `t` déclaré ainsi :

```
float t[3] [4] ;
```

Écrire les (seules) instructions permettant de calculer, dans une variable nommée `som`, la somme des éléments de `t` :

a. en utilisant le « formalisme usuel des tableaux à deux indices » ;

b. en utilisant le « formalisme pointeur ».

Solutions

a. La première solution ne pose aucun problème particulier :

```
int i, j ;
som = 0 ;
for (i=0 ; i<3 ; i++)
   for (j=0 ; j<4 ; j++)
      som += t[i] [j] ;
```

b. Le formalisme pointeur est ici moins facile à appliquer que dans le cas des tableaux à un indice. En effet, avec, par exemple, `float t[4]`, `t` est de

type `int *` et il correspond à un pointeur sur le premier élément du tableau. Il suffit donc d'incrémenter convenablement `t` pour parcourir tous les éléments du tableau.

En revanche, avec notre tableau `float t [3] [4]`, `t` est du type pointeur sur des **tableaux de 4 flottants** (type : `* float[4]`). La notation `*(t+i)` est généralement inutilisable sous cette forme puisque, d'une part, elle correspond à des valeurs de tableaux de 4 flottants et que, d'autre part, l'incrément `i` porte, non plus sur des flottants, mais sur des blocs de 4 flottants ; par exemple, `t+2` représente l'adresse du huitième flottant, compté à partir de celui d'adresse `t`.

Une solution consiste à « convertir » la valeur de `t` en un pointeur de type `float *`. On pourrait se contenter de procéder ainsi :

```
float * adt ;
   .....
adt = t ;
```

En effet, dans ce cas, l'affectation entraîne une conversion forcée de `t` en `float *`, ce qui ne change pas l'adresse correspondante (seule la nature du pointeur a changé).

Remarque

Cela n'est vrai que parce que l'on passe de pointeurs sur des groupes d'éléments à un pointeur sur ces éléments. Autrement dit, aucune contrainte d'alignement ne risque de nuire ici. Il n'en irait pas de même, par exemple, pour des conversions de `char *` en `int *`.

Généralement, on y gagnera en lisibilité en explicitant la conversion mise en œuvre à l'aide de l'opérateur de `cast`. Notez que, par ailleurs, cela peut éviter certains messages d'avertissement *(warnings)* de la part du compilateur.

Voici finalement ce que pourraient être les instructions demandées :

```
int i ;
int * adt ;
som = 0 ;
```

```
adt = (float *) t ;
for (i=0 ; i<12 ; i++)
   som += * (adt+i);
```

Exercice 39

Énoncé

Écrire une fonction qui fournit en valeur de retour la somme des éléments d'un tableau de flottants transmis, ainsi que sa dimension, en argument.

Écrire un petit programme d'essai.

Solution

En C++, par défaut, les arguments sont transmis par valeur. Mais, dans le cas d'un tableau, cette valeur, de type pointeur, n'est rien d'autre que son adresse. Quant à la transmission par référence, elle n'a pas de signification dans ce cas. Nous n'avons donc aucun choix en ce qui concerne le mode de transmission de notre tableau.

En ce qui concerne le nombre d'éléments (de type `int`), nous le transmettrons classiquement par valeur. L'en-tête de notre fonction pourra se présenter sous l'une des formes suivantes :

```
float somme (float t[], int n)
float somme (float * t, int n)
float somme (float t[5], int n) // déconseillé, car laisse
croire que t
                                // est de dimension fixe 5
```

En effet, la dimension réelle de `t` n'a aucune incidence sur les instructions de la fonction elle-même (elle n'intervient pas dans le calcul de l'adresse d'un élément du tableau[1] et elle ne sert pas à « allouer » un emplacement puisque le tableau en question aura été alloué dans la fonction appelant `somme`).

Voici ce que pourrait être la fonction demandée :

```
float somme (float t[], int n) // on peut écrire somme (float
```

```
* t,...
                                        // ou encore somme (float
t[4],...
                                        // mais pas  somme (float
t[n],...
{   int i ;
    float s = 0 ;
    for (i=0 ; i<n ; i++)
        s += t[i] ;              // on pourrait écrire s += *
(t+i) ;
    return s ;
}
```

Pour ce qui est du programme d'utilisation de la fonction `somme`, on peut, là encore, écrire le « prototype » sous différentes formes :

```
float somme (float [], int ) ;
float somme (float * , int ) ;
float  somme  (float  [5],  int  )  ;//  déconseillé  car  laisse
croire que t
                                         // est de dimension fixe
5
```

Voici un exemple d'un tel programme :

```
#include <iostream>
using namespace std ;
main()
{
   float somme (float *, int) ;
   float t[4] = {3, 2.5, 5.1, 3.5} ;
   cout << "somme de t : " << somme (t, 4) ;
}
```

Exercice 40

Énoncé

Écrire une fonction qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers (à un indice) de taille quelconque. On prévoira 4 arguments : le tableau, sa dimension, le maximum et le minimum. Pour chacun d'entre eux, on choisira le mode de

transmission le plus approprié (par valeur ou par référence). Dans le cas où la transmission par référence est nécessaire, proposer deux solutions : l'une utilisant effectivement cette notion de référence, l'autre la « simulant » à l'aide de pointeurs.

Écrire un petit programme d'essai.

Solution

En C++, par défaut, les arguments sont transmis par valeur. Mais, dans le cas d'un tableau, cette valeur, de type pointeur, n'est rien d'autre que l'adresse du tableau. Quant à la transmission par référence, elle n'a pas de signification dans ce cas. Nous n'avons donc aucun choix concernant le mode de transmission de notre tableau.

En ce qui concerne le nombre d'éléments du tableau, on peut indifféremment en transmettre l'adresse (sous forme d'un pointeur de type `int *`), ou la valeur ; ici, la seconde solution est la plus appropriée, puisque la fonction n'a pas besoin d'en modifier la valeur.

En revanche, en ce qui concerne le maximum et le minimum, ils ne peuvent pas être transmis par valeur, puisqu'ils doivent précisément être déterminés par la fonction. Il faut donc obligatoirement prévoir de passer :

- soit des références. L'en-tête de notre fonction se présentera ainsi :

```
void maxmin (int t[], int n, int & admax, int & admin)
```

- soit des pointeurs sur des `float`. L'en-tête de notre fonction se présentera ainsi :

```
void maxmin (int t[], int n, int * admax, int * admin)
```

L'algorithme de recherche de maximum et de minimum peut être calqué sur celui de l'exercice 39, en remplaçant `max` par `*admax` et `min` par `*admin`. Voici ce que pourrait être notre fonction :

- avec transmission par référence :

```
void maxmin (int t[], int n, int & admax, int & admin)
{   int i ;
    admax = t[1] ;
    admin = t[1] ;
```

```
    for (i=1 ; i<n ; i++)
       { if (t[i] > admax) admax = t[i] ;
         if (t[i] < admin) admin = t[i] ;
       }
}
```

- avec transmission par pointeurs

```
void maxmin (int t[], int n, int * admax, int * admin)
{   int i ;
    *admax = t[1] ;
    *admin = t[1] ;
    for (i=1 ; i<n ; i++)
       { if (t[i] > *admax) *admax = t[i] ;
         if (t[i] < *admin) *admin = t[i] ;
       }
}
```

Ici, si l'on souhaite éviter les « indirections » qui apparaissent systématiquement dans les instructions de comparaison, on peut travailler temporairement sur des variables locales à la fonction (nommées ici `max` et `min`). Cela nous conduit à une fonction de la forme suivante :

```
void maxmin (int t[], int n, int * admax, int * admin)
{  int i, max, min ;
   max = t[1] ;
   min = t[1] ;
   for (i=1 ; i<n ; i++)
      {  if (t[i] > max) max = t[i] ;
         if (t[i] < min) min = t[i] ;
      }
   *admax = max ;
   *admin = min ;
}
```

Voici un petit exemple de programme d'utilisation de la première fonction :

```
#include <iostream>
using namespace std ;
main()
{  void maxmin (int [], int, int &, int &) ;
   int t[8] = { 2, 5, 7, 2, 9, 3, 9, 4} ;
   int max, min ;
   maxmin (t, 8, max, min) ;
   cout << "valeur maxi : " << max << "\n" ;
   cout << "valeur mini : " << min << "\n" ;
}
```

Et voici le même exemple utilisant la seconde fonction :

```
#include <iostream>
using namespace std ;
main()
{  void maxmin (int [], int, int *, int *) ;
   int t[8] = { 2, 5, 7, 2, 9, 3, 9, 4} ;
   int max, min ;
   maxmin (t, 8, &max, &min) ;
   cout << "valeur maxi : " << max << "\n" ;
   cout << "valeur mini : " << min << "\n" ;
}
```

Exercice 41

Énoncé

Écrire une fonction qui fournit en retour la somme des valeurs d'un tableau de flottants à deux indices dont les dimensions sont fournies en argument.

Solution

Par analogie avec ce que nous avions fait dans l'exercice 39, nous pourrions songer à déclarer le tableau concerné dans l'en-tête de la fonction sous la forme `t[][]`. Mais cela n'est plus possible car, cette fois, pour déterminer l'adresse d'un élément `t[i][j]` d'un tel tableau, le compilateur doit en connaître la deuxième dimension.

Une solution consiste à considérer qu'on reçoit un pointeur (de type `float*`) sur le début du tableau et d'en parcourir tous les éléments (au nombre de `n*p` si `n` et `p` désignent les dimensions du tableau) comme si l'on avait affaire à un tableau à une dimension.

Cela nous conduit à cette fonction :

```
float somme (float * adt, int n, int p)
{  int i ;
   float s ;
    for (i=0 ; i<n*p ; i++)  s += adt[i] ;      /* ou s += *
(adt+i) */
```

```
   return s ;
}
```

Pour utiliser une telle fonction, la seule difficulté consiste à lui transmettre effectivement l'adresse de début du tableau, sous la forme d'un pointeur de type `int *`. Or, avec, par exemple `t[3][4]`, `t`, s'il correspond bien à la bonne adresse, est du type « pointeur sur des tableaux de 4 flottants ». A priori, toutefois, compte tenu de la présence du prototype, la conversion voulue sera mise en œuvre automatiquement par le compilateur. Toutefois, comme nous l'avons déjà dit dans l'exercice 38, on gagnera en lisibilité (et en éventuels messages d'avertissement !) en faisant appel à l'opérateur de « `cast` ».

Voici finalement un exemple d'un tel programme d'utilisation de notre fonction :

```
#include <iostream>

using namespace std ;
main()
{  float somme (float *, int, int) ;
   float t[3] [4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} } ;
   cout << "somme : " << somme ((float *)t, 3, 4) << "\n" ;
}
```

Exercice 42

Énoncé

Écrire un programme allouant dynamiquement un emplacement pour un tableau d'entiers, dont la taille est fournie en donnée. Utiliser ce tableau pour y placer des nombres entiers lus également en donnée. Créer ensuite dynamiquement un nouveau tableau destiné à recevoir les carrés des nombres contenus dans le premier. Supprimer le premier tableau, afficher les valeurs du second et supprimer le tout. On ne cherchera pas à traiter un éventuel problème de manque de mémoire.

Solution

Il nous faut utiliser deux variables (`adt1` et `adt2`) de type pointeur sur des entiers pour conserver les adresses des emplacements alloués pour chacun des deux tableaux d'entiers.

```
#include <iostream>
using namespace std ;
main()
{ int nval ;            //  nombre de valeurs
  int * adt1, * adt2 ;  // attention, pas int * adt1, adt2
  do { cout << "combien de valeurs : " ;
       cin >> nval ;
     }
  while (nval <= 0) ;    // on refuse les valeurs négatives
      /* allocation premier tableau, lecture valeurs */
  adt1 = new int [nval] ;
  cout << "donnez " << nval << " valeurs \n ";
  for (int i = 0 ; i<nval ; i++) cin >> adt1[i] ;   // ou cin
* (adt1+i)
     /* allocation second tableau, calcul des carrés */
  adt2 = new int [nval] ;
  for (int i = 0 ; i<nval ; i++) adt2[i] = adt1[i] * adt1[i] ;
     /* suppression premier tableau, affichage valeurs */
  delete [] adt1 ;
  cout << "voici leurs carrés : \n" ;
  for (int i = 0 ; i<nval ; i++) cout << adt2[i] << " " ;
     /* suppression du second tableau */
  delete [] adt2 ;
}
```

Voici un exemple d'exécution de ce programme :

```
combien de valeurs : 0
combien de valeurs : -2
combien de valeurs : 8
donnez 8 valeurs
1 2 5 9 7 3 0 8 6 -2
voici leurs carrés :
1 4 25 81 49 9 0 64
```

Exercice 43

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{
   char * ad1 ;
   ad1 = "bonjour" ;
   cout << ad1 << "\n" ;
   ad1 = "monsieur" ;
   cout << ad1 ;
}
```

Solution

L'instruction `ad1 = "bonjour"` place dans la variable `ad1` l'adresse de la chaîne constante `"bonjour"`. L'instruction `cout << ad1` se contente d'afficher la valeur de la chaîne dont l'adresse figure dans `ad1`, c'est-à-dire en l'occurrence `"bonjour"`. De manière comparable, l'instruction `ad1 = "monsieur"` place l'adresse de la chaîne constante `"monsieur"` dans `ad1` ; l'instruction `cout << ad1` affiche la valeur de la chaîne ayant maintenant l'adresse contenue dans `ad1`, c'est-à-dire `"monsieur"`.

Finalement, ce programme affiche tout simplement :

```
bonjour
monsieur
```

On aurait obtenu plus simplement le même résultat en écrivant :

```
cout << "bonjour\nmonsieur" ;
```

Exercice 44

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
```

```
{
   char * adr = "bonjour" ;                    /* 1 */
   int i ;
   for (i=0 ; i<3 ; i++) cout << adr[i] ;      /* 2 */
   cout << "\n" ;
   i = 0 ;
   while (adr[i]) cout << adr[i++] ;           /* 3 */
}
```

Solution

La déclaration `/* 1 */` place dans la variable `adr` l'adresse de la chaîne constante `bonjour`. L'instruction `/* 2 */` affiche les caractères `adr[0]`, `adr[1]` et `adr[2]`, c'est-à-dire les 3 premiers caractères de cette chaîne. L'instruction `/* 3 */` affiche tous les caractères à partir de celui d'adresse `adr`, tant que l'on a pas affaire à un caractère nul ; comme notre chaîne `"bonjour"` est précisément terminée par un tel caractère nul, cette instruction affiche finalement, un par un, tous les caractères de `"bonjour"`.

En définitive, le programme fournit simplement les résultats suivants :

```
bon
bonjour
```

Exercice 45

Énoncé

Écrire le programme précédent (exercice 44), **sans utiliser le « formalisme tableau »** (il existe plusieurs solutions).

Solutions

Voici deux solutions possibles :

a. On peut remplacer systématiquement la notation `adr[i]` par `*(adr+i)`, ce qui conduit à ce programme :

```
#include <iostream>
```

```
using namespace std ;
main()
{
   char * adr = "bonjour" ;
   int i ;
   for (i=0 ; i<3 ; i++) cout << *(adr+i) ;
   cout << "\n" ;
   i = 0 ;
   while (adr[i]) cout <<  *(adr+i++) ;
}
```

b. On peut également parcourir notre chaîne, non plus à l'aide d'un « indice » `i`, mais en incrémentant un pointeur de type `char * ` : il pourrait s'agir tout simplement de `adr`, mais généralement on préférera ne pas détruire cette information et en employer une copie :

```
#include <iostream>
using namespace std ;
main()
{
   char * adr = "bonjour" ;
   char * adb ;
   for (adb=adr ; adb<adr+3 ; adb++) cout << *adb ;
   cout << "\n" ;
   adb = adr ;
   while (*adb) cout <<  *(adb++) ;
}
```

Notez bien que si nous incrémentions directement `adr` dans la première instruction d'affichage, nous ne disposerions plus de la « bonne adresse » pour la seconce instruction d'affichage.

Exercice 46

Énoncé

Écrire un programme qui demande à l'utilisateur de lui fournir un nombre entier entre `1` et `7` et qui affiche le nom du jour de la semaine ayant le numéro indiqué (`lundi` pour `1`, `mardi` pour `2`, ... `dimanche` pour `7`).

Solution

Une démarche consiste à créer un « tableau de 7 pointeurs sur des chaînes », correspondant chacune au nom d'un jour de la semaine. Comme ces chaînes sont ici constantes, il est possible de créer un tel tableau par une déclaration comportant une initialisation de la forme :

```
char * jour [7] = { "lundi", "mardi", ...
```

N'oubliez pas alors que `jour[0]` contiendra l'adresse de la première chaîne, c'est-à-dire l'adresse de la chaîne constante `"lundi"` ; `jour[1]` contiendra l'adresse de `"mardi"`...

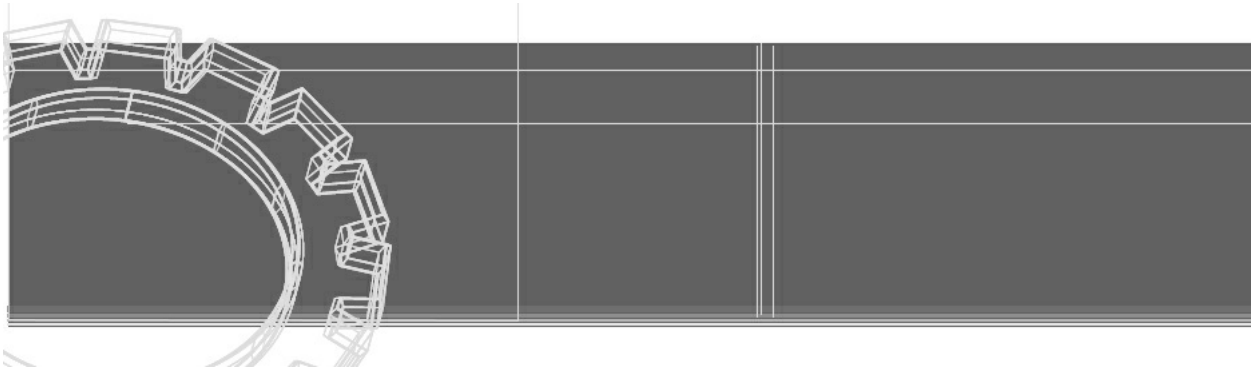
Pour afficher la valeur de la chaîne de rang `i`, il suffit de remarquer que son adresse est simplement `jour[i-1]`.

D'où le programme demandé :

```
#include <iostream>
using namespace std ;
main()
{
    char * jour [7] = { "lundi",     "mardi",   "mercredi",
"jeudi",
                        "vendredi", "samedi", "dimanche"
                      } ;
   int i ;
   do
      { cout << "donnez un nombre entier entre 1 et 7 : " ;
        cin >> i ;
      }
   while ( i<=0 || i>7) ;
   cout << "le jour numéro " << i << " de la semaine est " <<
jour[i-1] ;
}
```

1. Il n'en irait pas de même pour des tableaux à plusieurs indices.

Chapitre 5
Les structures



Rappels

Déclaration d'un type structure et des variables de ce type

C++ permet de déclarer un type structure, de cette manière :

```
struct enreg {  int numero ;
                int qte ;
                float prix ;
             } ;
```

Cette déclaration définit un **type (modèle) de structure** mais ne réserve pas de variables correspondant à cette structure. Ce type s'appelle ici `enreg` et il précise le nom et le type de chacun des champs constituant la structure (`numero`, `qte` et `prix`).

Une fois un tel type de structure défini, nous pouvons déclarer des variables du type correspondant. Par exemple, l'instruction :

```
enreg art1, art2 ;
```

réserve deux emplacements nommés `art1` et `art2`, « de type `enreg` », destinés à contenir chacun deux entiers et un flottant.

Les champs d'une structure peuvent être de n'importe quel type de base, d'un type tableau ou structure. De même, les éléments d'un tableau peuvent être d'un type structure.

Utilisation d'une (variable de type) structure

Un champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. On désigne un champ donné en faisant suivre le nom de la variable structure de l'opérateur « point » (.), suivi du nom de champ, comme dans ces exemples utilisant les déclarations précédentes :

```
art1.numero    // champ numero de la structure art1
art2.prix      // champ prix de la structure art2
```

Contrairement au tableau, une variable de type structure peut être affectée à une variable de même type (même nom de modèle).

Initialisation des structures

Les structures de classe automatique ne sont pas initialisées par défaut. Celles de classe statique voient leurs champs initialisés à « zéro » (entier nul, flottant nul, caractère de code nul, pointeur nul). En toute rigueur, cette règle s'applique aux champs qui sont des scalaires ou des tableaux de scalaires. Si certains champs sont eux-mêmes des structures, la règle s'appliquera à chacun de leurs champs, et ainsi de suite.

À l'instar d'un tableau, une structure peut être initialisée lors de sa déclaration, comme dans cette instruction qui utilise le type `enreg` défini précédemment :

```
enreg art1 = { 100, 285, 200 } ; // = facultatif depuis C++11.
```

On peut omettre certaines valeurs.

Exercice 47

Énoncé

Soit le modèle (type) de structure suivant :

```
struct point
{  char c ;
   int x, y ;
} ;
```

Écrire une fonction qui reçoit en argument une structure de type `point` et qui en affiche le contenu sous la forme :

```
point B de coordonnées 10 12
```

a. en transmettant en argument la **valeur** de la structure concernée ;

b. en transmettant en argument l'**adresse** de la structure concernée ;

c. en transmettant la structure concernée par **référence**.

Dans les trois cas, on écrira un petit programme d'essai de la fonction ainsi réalisée.

Solutions

a. Voici la fonction demandée :

```
void affiche (point p)
{   cout << "point " << p.c << " de coordonnées "
         << p.x << " " <<p.y << "\n" ;
}
```

Notez que sa compilation nécessite obligatoirement la déclaration du type `point`.

Voici un petit programme complet qui affecte les valeurs `'A'`, `10` et `12` aux différents champs d'une structure nommée `s`, avant d'en afficher les valeurs à l'aide de la fonction précédente :

```
#include <iostream>
using namespace std ;

struct point
{  char c ;
   int x, y ;
} ;
```

```
void affiche (point p)
{   cout << "point " << p.c << " de coordonnées "
         << p.x << " " <<p.y << "\n" ;
}
main()
{   void affiche (point) ;   // déclaration de affiche
    point s ;
    s.c = 'A' ;
    s.x = 10 ;
    s.y = 12 ;
    affiche (s) ;
}
```

b. Voici la deuxième fonction demandée, accompagnée de son programme de test :

```
#include <iostream>
using namespace std ;
struct point
{  char c ;
   int x, y ;
} ;
void affiche (point * adp)
{   cout << "point " << adp->c << " de coordonnées "
         << adp->x << " " << adp->y ;
}
main()
{   void affiche (point *) ;
    point s ;
    s.c = 'A' ;
    s.x = 10 ;
    s.y = 12 ;
    affiche (&s) ;
}
```

Notez que l'on doit, cette fois, faire appel à l'opérateur `->`, à la place de l'opérateur « point » (`.`), puisque l'on travaille avec un pointeur sur une structure, et non plus avec la valeur de la structure elle-même. Toutefois l'usage de `->` n'est pas totalement indispensable, dans la mesure où, par exemple, `adp->x` est équivalent à `(*adp).x`.

c. Voici la troisième fonction demandée, accompagnée de son programme de test :

```
#include <iostream>
using namespace std ;
```

```
struct point
{  char c ;
   int x, y ;
} ;
void affiche (point & p)
{   cout << "point " << p.c << " de coordonnées "
         << p.x << " " << p.y ;
}
main()
{   void affiche (point &) ;
    point s ;
    s.c = 'A' ;
    s.x = 10 ;
    s.y = 12 ;
    affiche (s) ;
}
```

Remarque

Au lieu d'affecter des valeurs aux champs `c`, `x` et `y` de notre structure `s` (dans les trois programmes d'essai), nous pourrions (ici) utiliser les possibilités d'initialisation offertes par le langage C, en écrivant :

```
point s = {'A', 10, 12} ;
```

Exercice 48

Énoncé

Soit le type structure `enreg` défini ainsi :

```
const int NMOIS = 12 ;
struct enreg
{  int stock ;
   float prix ;
   int ventes [NMOIS]
}
```

Écrire une fonction nommée `raz` qui « met à zéro » les champs `stock` et

ventes d'une structure de ce type, transmise en argument. La fonction ne comportera pas de valeur de retour.

Écrire un petit programme d'essai qui affecte tout d'abord des valeurs aux différents champs d'une telle structure, avant de leur appliquer la fonction raz. On affichera les valeurs de la structure, avant et après appel (on pourra s'aider d'une fonction d'affichage).

Solution

Ici, pour que la fonction puisse modifier la valeur d'une structure reçue en argument, il est nécessaire qu'elle en reçoive, soit la référence, soit l'adresse. Voici un exemple complet utilisant la première possibilité :

```
#include <iostream>
using namespace std ;
const int NMOIS = 12 ;
struct enreg
{  int stock ;
   float prix ;
   int ventes [NMOIS] ;
} ;
void raz (enreg & s)
{ s.stock = 0 ;
  for (int i=0 ; i<NMOIS ; i++)
    s.ventes[i] = 0 ;
  return ;
}
void affiche (enreg  s)   // transmission par valeur ici
{ cout << "stock : " << s.stock << "\n" ;
  cout << "prix :  " << s.prix << "\n" ;
  cout << "ventes : " ;
   for (int i = 0 ; i<NMOIS ; i++)  cout << s.ventes[i] << "
" ;
  cout << "\n" ;
}

main()
{ void raz (enreg &) ;
  enreg e = {12, 5.25, {12, 23, 4, 8, 4, 9, 5, 2, 7, 2, 8, 7}
} ;
  cout << "contenu avant raz :\n" ;
  affiche (e) ;
  raz (e) ;
```

```
  cout << "contenu apres raz :\n" ;
  affiche (e) ;
}
```

Voici un exemple d'exécution de ce programme :

```
contenu avant raz :
stock : 12
prix :  5.25
ventes : 12 23 4 8 4 9 5 2 7 2 8 7
contenu apres raz :
stock : 0
prix :  5.25
ventes : 0 0 0 0 0 0 0 0 0 0 0 0
```

À titre indicatif, voici ce que serait notre fonction `raz`, avec une transmission par pointeur :

```
void raz (enreg * ads)
{ ads->stock = 0 ;
  for (int i=0 ; i<NMOIS ; i++)
    ads->ventes[i] = 0 ;
  return ;
}
```

Dans la fonction `main`, sa déclaration et son appel deviendraient :

```
void raz (enreg *) ;
raz (&e)
```

Exercice 49

Énoncé

Soit le type structure suivant, représentant un point d'un plan :

```
struct point { char c ;     // nom attribué au point
               int x, y ;   // ses coordonnées
             }
```

Écrire une fonction qui reçoit en argument l'adresse d'une structure du type `point` et qui renvoie en résultat une structure de même type correspondant à un point de même nom et de coordonnées opposées.

Écrire un petit programme d'essai.

Solution

Voici ce que pourrait être notre fonction (nous avons reproduit la déclaration de `point`, laquelle pourrait éventuellement figurer dans un fichier en-tête séparé qu'on incorporerait par une directive `#include`) :

```
#include <iostream>
using namespace std ;
struct point
   { char c ;
     int x, y ;
   } ;
point sym (point * adp)
{  point res ;
   res.c = adp->c ;
   res.x = - adp->x ;
   res.y = - adp->y ;
   return res ;
}
```

Notez la « dissymétrie » d'instructions telles que `res.c = adp->c` ; on y fait appel à l'opérateur « . » à gauche et à l'opérateur « `->` » à droite (on pourrait cependant écrire `res.c = (*adp).c`).

Voici un exemple d'essai de notre fonction (ici, nous avons utilisé les possibilités d'initialisation d'une structure pour donner des valeurs à `p1`) :

```
main()
{  point sym (point *) ;
   point p1 = {'P', 5, 8} ;
   point p2 ;
   p2 = sym (&p1) ;
   cout << p1.c << " " << p1.x << " " << p1.y << "\n" ;
   cout << p2.c << " " << p2.x << " " << p2.y << "\n" ;
}
```

Remarque

Si l'énoncé ne l'avait pas imposé, nous aurions pu transmettre par référence l'argument de la fonction `sym`. Nous aurions pu également

utiliser une transmission par valeur, ce qui n'aurait guère été pénalisant pour une information de si petite taille. En théorie, ce choix du mode de transmission (valeur, référence ou adresse) existe également pour la valeur de retour. Toutefois, cette dernière est généralement (comme c'est le cas ici) créée dans une variable locale à la fonction. Dans ces conditions, en transmettre l'adresse ou la référence reviendrait à renvoyer l'adresse de quelque chose destiné à disparaître. En toute rigueur, on pourrait renvoyer l'adresse d'un emplacement alloué dynamiquement, mais encore faudrait-il définir clairement à qui incomberait la responsabilité de sa suppression ultérieure.

Exercice 50

Énoncé

Soit la structure suivante, représentant un point d'un plan :

```
struct point
   { char c ;       // nom du point
     int x, y ;     // coordonnées
   } ;
```

1. Écrire la déclaration d'un tableau (nommé `courbe`) de `NP` points (`NP` supposé défini par une constante).
2. Écrire une fonction (nommée `affiche`) qui affiche les valeurs des différents « points » du tableau `courbe`, transmis en argument, sous la forme :

   ```
   point D de coordonnées 10 2
   ```

3. Écrire un programme qui :
   - lit en données des valeurs pour le tableau `courbe ;`
   - fait appel à la fonction précédente pour les afficher.

Solutions

1. Il suffit de déclarer un tableau de structures :

```
struct point courbe [NP] ;
```

2. Comme `courbe` est un tableau, on ne peut qu'en transmettre l'adresse en argument de `affiche`. Il est préférable de prévoir également en argument le nombre de points. Voici ce que pourrait être notre fonction :

```
void affiche (point courbe [], int np)
      /* courbe : adresse de la première structure du tableau
*/
     /* (on pourrait écrire point * courbe)
*/
     /* np : nombre de points de la courbe
 */
{  int i ;
   for (i=0 ; i<np ; i++)
      cout << "point " << courbe[i].c << " de coordonnées "
           << courbe[i].x << " " << courbe[i].x << "\n" ;
}
```

Comme pour n'importe quel tableau à une dimension transmis en argument, il est possible de ne pas en mentionner la dimension dans l'en-tête de la fonction. Bien entendu, comme, en fait, l'identificateur `courbe` n'est qu'un pointeur de type `point *` (pointeur sur la première structure du tableau), nous aurions pu également écrire `point * courbe`.

Notez que, comme à l'accoutumée, le « formalisme tableau » et le « formalisme pointeur » peuvent être indifféremment utilisés (voire combinés). Par exemple, notre fonction aurait pu également s'écrire :

```
void affiche (point * courbe, int np)
{  point * adp ;
   int i ;
   for (i=0, adp=courbe ; i<np ; i++, adp++)
      cout << "point " << (courbe+i)-> c << " de coordonnées "
           << (courbe+i)->x << (courbe+i)->y) ;
}
```

3. Voici ce que pourrait donner le programme demandé :

```
#include <iostream>
using namespace std ;
const int NP = 4 ; // nombre de points d'une courbe
struct point
   { char c ;
     int x, y ;
   } ;
```

```
void affiche (point courbe [], int np)
{
   int i ;
   for (i=0 ; i<np ; i++)
      cout << "point " << courbe[i].c << " de coordonnées "
           << courbe[i].x << " " << courbe[i].x << "\n" ;
}
main()
{  point courbe [NP] ;
   int i ;
   void affiche (point [], int) ;
        /* lecture des différents points de la courbe */
   for (i=0 ; i<NP ; i++)
     {  cout << "nom (1 caractère) et coordonnées point " <<
i+1 << "\n" ;
       cin >> courbe[i].c >> courbe[i].x >> courbe[i].y ;
    }
    affiche (courbe, NP) ;
}
```

Exercice 51

Énoncé

Écrire le programme de la question 3 de l'exercice précédent, **sans utiliser de structures**. On prévoira toujours une fonction pour lire les informations relatives à un point.

Solution

Ici, il nous faut obligatoirement prévoir 3 tableaux différents de même taille : un pour les noms de points, un pour leurs abscisses et un pour leurs ordonnées. Le programme ne présente pas de difficultés particulières (son principal intérêt est de pouvoir être comparé au précédent !).

```
#include <iostream>
using namespace std ;

const int NP = 4 ;            // nombre de points d'une courbe
main()
```

```
{  char c [NP] ;              // noms des différents points
    int  x [NP] ;                  // abscisses des différents
points
    int y  [NP] ;                  // ordonnées des différents
points
   int i ;

   void affiche (char [], int[], int[], int) ;
        /* lecture des différents points de la courbe */
   for (i=0 ; i<NP ; i++)
   {  cout << "nom (1 caractère) et coordonnées point "
           <<  i+1 << " :\n" ;
      cin >> c[i] >> x[i] >> y[i] ;
   }
   affiche (c, x, y, NP) ;
}

void affiche (char c[], int x[], int y[], int np)
{    for (int i=0 ; i<np ; i++)
       cout << "point " << c[i] << " de coordonnées "
            << x[i] << " " <<  y[i] << "\n";
}
```

Exercice 52

Énoncé

Soient les deux modèles de structure `date` et `personne` déclarés ainsi :

```
cont int LG_NOM = 30 ;
struct date
    { int jour ;
      int mois ;
      int annee ;
    } ;
struct personne
 { char nom [LG_NOM+1] ;  // chaîne de caractères (de style
C)
                          // représentant le nom
   struct date date_embauche ;
   struct date date_poste ;
 } ;
```

Écrire une fonction qui reçoit en argument une structure de type `personne` et qui en remplit les différents champs avec un dialogue se présentant sous l'une des 2 formes suivantes :

```
nom : DUPONT
date embauche (jj mm aa) : 16 1 75
date poste = date embauche ? (O/N) : O

nom : DUPONT
date embauche (jj mm aa) : 10 3 81
date poste = date embauche ? (O/N) : N
date poste (jj mm aa) : 23 8 91
```

Notre fonction doit modifier le contenu d'une structure de type `personne` ; il est donc nécessaire qu'elle en reçoive la référence ou l'adresse en argument. Ici, l'énoncé n'imposant rien de particulier, nous choisirons une transmission par référence. Voici ce que pourrait être la fonction demandée :

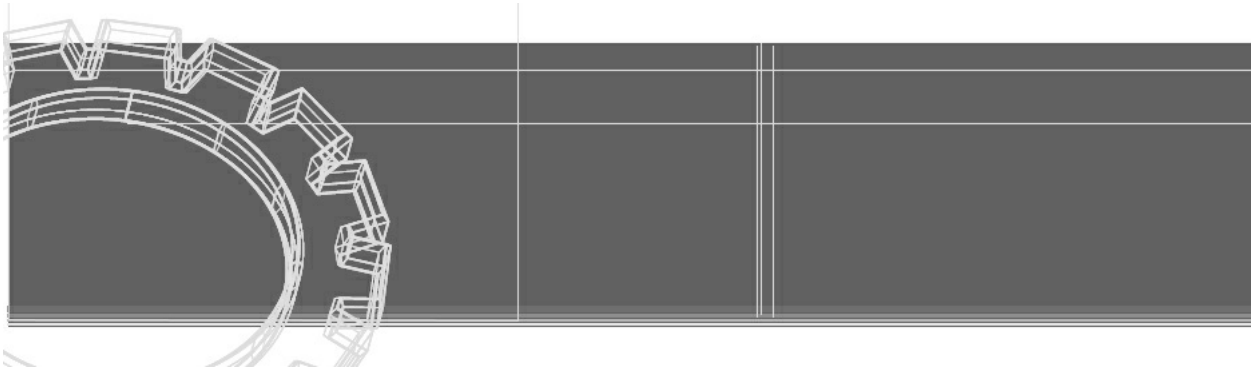
```
void remplit (personne & p)
{   char rep ;           // pour lire une réponse de type O/N
    cout << "nom : " ;
     cin >> p.nom ;         // attention, pas de contrôle de
longueur

    cout << "date embauche (jj mm aa) : " ;
    cin >> p.date_embauche.jour
        >> p.date_embauche.mois
        >> p.date_embauche.annee ;

    cout << "date poste = date embauche ? (O/N) : " ;
    cin >> rep ;
    if (rep == 'O') p.date_poste = p.date_embauche ;
       else  { cout << "date poste (jj mm aa) : " ;
               cin >> p.date_poste.jour
                   >> p.date_poste.mois
                   >> p.date_poste.annee ;
             }
}
```

Voici, à titre indicatif, un petit programme d'essai de notre fonction (sa compilation nécessite les déclarations des structures `date` et `personne`) :

```
main()
```

```
{  void remplit (personne &) ; // déclaration remplit
   personne bloc ;
   remplit (bloc) ;
   cout << "nom : " << bloc.nom << "\ndate embauche : "
        << bloc.date_embauche.jour << " "
        << bloc.date_embauche.mois << " "
        << bloc.date_embauche.annee << "\n"
        <<"date poste  : "
        << bloc.date_poste.jour << " "
        << bloc.date_poste.mois << " "
        << bloc.date_poste.annee  ;
}
```

Chapitre 6
De C à C++



N.B. Ce chapitre constitue le « point d'entrée » de l'ouvrage pour les programmeurs abordant l'étude de C++, en ayant déjà une connaissance du langage C. Les autres programmeurs pourront continuer leur étude en ignorant les rappels faisant un parallèle entre C et C++ et les exercices 53, 54 et 55.

Rappels

C++ est presque un sur-ensemble du C, tel qu'il est défini par la norme ANSI. Seules quelques incompatibilités existent ; nous rappelons ici les principales. Par ailleurs, C++ dispose, par rapport au C ANSI, d'un certain nombre de spécificités qui ne sont pas véritablement axées sur la programmation orientée objet. Elles sont également examinées ici.

Déclarations de fonctions

En C++, toute fonction utilisée dans un fichier source doit obligatoirement

avoir fait l'objet :

- soit d'une déclaration sous la forme d'un **prototype** (il précise à la fois le nom de la fonction, le type de ses arguments éventuels et le type de sa valeur de retour), comme dans cet exemple :

```
float fexp (int, double, char *) ;
```

- soit d'une définition préalable au sein du même fichier source (ce dernier cas étant d'ailleurs peu conseillé, dans la mesure où des problèmes risquent d'apparaître dès lors qu'on sépare ladite fonction du fichier source en question).

En C, une fonction pouvait ne pas être déclarée (auquel cas on considérait, par défaut, que sa valeur de retour était de type `int`), ou encore déclarée partiellement (sans fournir le type de ses arguments), comme dans :

```
float fexp () ;
```

Fonctions sans arguments

En C++, une fonction sans argument se définit (au niveau de l'en-tête) et se déclare (au niveau du prototype) en fournissant une « liste d'arguments vide » comme dans :

```
float fct () ;
```

En C, on pouvait indifféremment utiliser cette notation ou faire appel au mot-clé `void`, comme dans :

```
float fct (void)
```

Fonctions sans valeur de retour

En C++, une fonction sans valeur de retour se définit (en-tête) et se déclare (prototype) obligatoirement à l'aide du mot-clé `void`, comme dans :

```
void fct (int, double) ;
```

En C, l'emploi du mot-clé `void` était, dans ce cas, facultatif.

Le qualificatif const

En C++, un symbole global déclaré avec le qualificatif `const` :

- a une portée limitée au fichier source concerné, tandis qu'en C il pouvait éventuellement être utilisé dans un autre fichier source (en utilisant le mot-clé `extern`) ;
- peut être utilisé dans une expression constante (calculable au moment de la compilation), alors qu'il ne pouvait pas l'être en C ; ce dernier point permet notamment d'utiliser de tels symboles pour définir la taille d'un tableau (en C, il fallait obligatoirement avoir recours à une définition de symboles par la directive `#define`).

Le type void *

En C++, un pointeur de type `void *` ne peut pas être converti implicitement lors d'une affectation en un pointeur d'un autre type ; la chose était permise en C. Bien entendu, en C++, il reste possible de faire appel à l'opérateur de `cast`.

Nouvelles possibilités d'entrées-sorties

C++ dispose de nouvelles facilités d'entrées-sorties. Bien qu'elles soient fortement liées à des aspects P.O.O. (surdéfinition d'opérateur en particulier), elles sont parfaitement utilisables en dehors de ce contexte. C'est tout particulièrement le cas des possibilités d'entrées-sorties conversationnelles (clavier, écran) qui remplacent avantageusement les fonctions `printf` et `scanf`. Ainsi :

cout << **expression**$_1$ << **expression**$_2$ << **.....** << **expression**$_n$

affiche sur le flot `cout` (connecté par défaut à la sortie standard `stdout`) les valeurs des différentes expressions indiquées, selon une présentation adaptée à leur type (on sait distinguer les attributs de signe et on peut afficher des valeurs de pointeurs). De même :

cin >> **lvalue**$_1$ >> **lvalue**$_2$ >> **.....** >> **lvalue**$_n$

lit sur le flot `cin` (connecté par défaut à l'entrée standard `stdin`) des informations de l'un des types `char`, `short`, `int`, `long`, `float`, `double` ou `char *` (on sait distinguer les attributs de signe ; en revanche, les pointeurs ne sont pas admis). Les conventions d'analyse des caractères lus sont comparables à celles de `scanf`, avec cette principale différence que la lecture

d'un caractère commence par sauter les espaces blancs (espace, tabulation horizontale, tabulation verticale, fin de ligne, changement de page).

Nouvelle forme de commentaires

Les deux caractères `//` permettent d'introduire des commentaires de fin de ligne : tout ce qui suit ces caractères, jusqu'à la fin de la ligne, est considéré comme un commentaire.

Emplacement libre des déclarations

En C++, il n'est plus nécessaire de regrouper les déclarations en début de fonction ou en début de bloc. Il devient ainsi possible d'employer des expressions dans des initialisations, comme dans cet exemple :

```
int n ;
   .....
n = ... ;
   .....
int q = 2*n - 1 ;
   .....
```

Ces possibilités s'appliquent également aux instructions structurées `for`, `switch`, `while` et `do...while`, comme dans cet exemple :

```
for (int i=0 ; ... ; ...)   // la portée de i est limitée au
bloc qui suit
{ .....
}
```

La transmission par référence

En faisant suivre du symbole `&` le type d'un argument dans l'en-tête (et dans le prototype) d'une fonction, on réalise une transmission par référence. Cela signifie que les éventuelles modifications effectuées au sein de la fonction porteront sur l'argument effectif de l'appel et non plus sur une copie. On notera qu'alors l'argument effectif doit obligatoirement être une `lvalue` du même type que l'argument muet correspondant. Toutefois, si l'argument muet est, de surcroît, déclaré avec l'attribut `const`, la fonction reçoit quand même une copie de l'argument effectif correspondant, lequel peut alors être une constante ou une expression d'un type susceptible d'être converti dans le

type attendu.

Ces possibilités de transmission par référence s'appliquent également à une valeur de retour, comme dans :

```
int & f() { ..... ]
```

Dans ce cas, la valeur de retour de `f` (si elle existe) sera transmise par référence. Il devra donc s'agir d'une `lvalue` modifiable : il ne peut donc plus s'agir d'une variable déclarée `const` (dite `lvalue` non modifiable), ni bien sûr d'une `rvalue` (expression ou constante littérale). De plus, elle devra continuer d'exister après la sortie de la fonction (ce qui exclut les variables locales à la fonction). Outre son utilisation classique dans une expression, l'appel de `f` peut également être utilisé comme une `lvalue` :

```
f() = .....
```

De façon un peu artificielle, il est possible de déclarer :

```
const int & f() { ..... ]
```

Dans ce cas, l'éventuelle expression `return` pourra mentionner, outre une `lvalue`, une constante ou une expression dont la fonction renverra alors une copie temporaire. On ne pourra plus utiliser `f()` comme une `lvalue`.

La notion de référence est théoriquement indépendante de celle de transmission d'argument ; en pratique, elle est rarement utilisée en dehors de ce contexte.

Les arguments par défaut

Dans la déclaration d'une fonction (prototype), il est possible de prévoir pour un ou plusieurs arguments (obligatoirement les derniers de la liste) des valeurs par défaut ; elles sont indiquées par le signe `=`, à la suite du type de l'argument comme dans cet exemple :

```
float fct (char, int = 10, float = 0.0) ;
```

Ces valeurs par défaut seront alors utilisées lorsqu'on appellera ladite fonction avec un nombre d'arguments inférieur à celui prévu. Par exemple,

avec la précédente déclaration, l'appel `fct ('a')` sera équivalent à `fct ('a', 10, 0.0)` ; de même, l'appel `fct ('x', 12)` sera équivalent à `fct ('x', 12, 0.0)`. En revanche, l'appel `fct ()` sera illégal.

Surdéfinition de fonctions

En C++, il est possible, au sein d'un même programme, que plusieurs fonctions possèdent le même nom. Dans ce cas, lorsque le compilateur rencontre l'appel d'une telle fonction, il effectue le choix de la « bonne » fonction en tenant compte de la nature des arguments effectifs. D'une manière générale, si les règles utilisées par le compilateur pour sa recherche sont assez intuitives, leur énoncé précis est assez complexe et nous ne le rappellerons pas ici (on le trouvera, par exemple, dans l'annexe de nos différents ouvrages consacrés à C++ et publiés également aux éditions Eyrolles.) Signalons simplement que ces règles peuvent faire intervenir toutes les conversions usuelles (promotions numériques et conversions standard, ces dernières pouvant être dégradantes), ainsi que les conversions définies par l'utilisateur en cas d'argument de type classe, à condition qu'aucune ambiguïté n'apparaisse.

Signalons également que le mode de transmission d'un argument (valeur ou référence) n'intervient pas dans le choix d'une fonction surdéfinie. En revanche, le qualificatif `const` appliqué à une référence permet bien, quant à lui, de distinguer deux fonctions différentes :

```
void f(int &) ;          // appelée pour une lvalue modifiable
void f(const int & ) ;   // appelée, après recopie de la valeur, pour une
                         // lvalue non modifiable (const) ou une expression
```

Gestion dynamique de la mémoire

Si `type` représente la description d'un type absolument quelconque, on peut allouer un emplacement pour un élément de type `type` par :

```
new type
```

On peut également allouer l'emplacement nécessaire pour ***n* éléments** du type `type` par :

```
new type [n]
```

Le résultat est un pointeur (de type `type *`) sur l'élément correspondant (ou le premier élément dans le deuxième cas). En cas d'échec d'allocation, la norme laisse l'implémentation libre de choisir entre le renvoi d'un pointeur de valeur nulle ou le déclenchement d'une exception `bad_alloc` (les exceptions font l'objet du chapitre 15).

Si `adr` désigne l'adresse de l'emplacement ainsi alloué, sa libération se fait à l'aide de l'une des deux syntaxes :

```
delete adr        // pour les éléments simples
delete [] adr     // pour les tableaux d'éléments
```

Les fonctions en ligne

Une fonction en ligne (on dit aussi « développée ») est une fonction dont les instructions sont incorporées par le compilateur (dans le module objet correspondant) à chaque appel. Cela évite la perte de temps nécessaire à un appel usuel (changement de contexte, copie des valeurs des arguments sur la « pile »...) ; en revanche, les instructions en question sont générées plusieurs fois.

Les fonctions « en ligne » offrent le même intérêt que les macros, sans présenter de risques d'effets de bord. Une fonction en ligne est nécessairement définie en même temps qu'elle est déclarée (elle ne peut plus être compilée séparément) et son en-tête est précédé du mot-clé `inline`, comme dans :

```
inline fct ( ...) {  ..... }
```

Exercice 53

Énoncé

Quelles erreurs seront détectées par un compilateur C++ dans ce fichier source qui est accepté par un compilateur C ?

```
main()
{
   int a=10, b=20, c ;
```

```
   c = g(a, b) ;
   printf ("valeur de g(%d,%d) = %d", a, b, c) ;
}
g(int x, int y)
{
   return (x*x + 2*x*y + y*y) ;
}
```

Solutions

1. La fonction `g` doit obligatoirement faire l'objet d'une déclaration (sous la forme d'un prototype) dans la fonction `main`. Par exemple, on pourrait introduire (n'importe où avant l'appel de `g`) :

```
int g (int, int) ;
```

ou encore :

```
int g (int x, int y) ;
```

Rappelons que, dans ce dernier cas, les noms `x` et `y` sont fictifs : ils n'ont aucun rôle dans la suite et ils n'interfèrent nullement avec d'autres variables de même nom qui pourraient être déclarées dans la même fonction (ici `main`).

2. La fonction `printf` doit, elle aussi, comme toutes les fonctions C++ (le compilateur n'étant pas en mesure de distinguer les fonctions de la bibliothèque des fonctions définies par l'utilisateur), faire l'objet d'un prototype. Naturellement, il n'est pas nécessaire de l'écrire explicitement ; il est obtenu par incorporation du fichier en-tête correspondant :

```
#include <cstdio>
```

Les symboles définis dans ce fichier appartiennent à l'espace de noms `std`, de sorte qu'il faut également prévoir une instruction :

```
using namespace std ;
```

Notez que certains compilateurs C refusent l'absence de prototype pour une fonction de la bibliothèque standard telle que `printf` (mais la norme ANSI n'imposait rien à ce sujet !).

Exercice 54

Énoncé

Écrire correctement en C ce programme qui est correct en C++ :

```
#include <cstdio>
using namespace std ;
const int nb = 10 ;
const int exclus = 5 ;
main()
{
   int valeurs [nb] ;
   int i, nbval = 0 ;
   printf ("donnez %d valeurs :\n", nb) ;
   for (i=0 ; i<nb ; i++) scanf ("%d", &valeurs[i]) ;
   for (i=0 ; i<nb ; i++)
      switch (valeurs[i])
       { case exclus-1 :
         case exclus   :
         case exclus+1 : nbval++ ;
       }
   printf ("%d valeurs sont interdites", nbval) ;
}
```

Solution

En C, les symboles `nb` et `exclus` ne sont pas utilisables dans des expressions constantes. Il faut donc les définir soit comme des variables, soit à l'aide d'une directive `#define` comme suit :

```
#include <stdio.h>
#define NB 10
#define EXCLUS 5
main()
{
   int valeurs [NB] ;
   int i ;
   int nbval=0 ;
   printf ("donnez %d valeurs :\n", NB) ;
   for (i=0 ; i<NB ; i++) scanf ("%d", &valeurs[i]) ;
   for (i=0 ; i<NB ; i++)
      switch (valeurs[i])
```

```
        { case EXCLUS-1 :
          case EXCLUS   :
          case EXCLUS+1 : nbval++ ;
        }
   printf ("%d valeurs sont interdites", nbval) ;
}
```

Exercice 55

Énoncé

Modifier le programme C suivant, de façon qu'il soit correct en C++ et qu'il ne fasse appel qu'aux nouvelles possibilités d'entrées-sorties de C++, c'est-à-dire qu'il évite les appels à `printf` et `scanf` :

```
#include <stdio.h>
main()
{
   int n ; float x ;
   printf ("donnez un entier et un flottant\n") ;
   scanf ("%d %e", &n, &x) ;
    printf ("le produit de %d par %e\n'est : %e", n, x,
n*x) ;
}
```

Solution

```
#include <iostream>
using namespace std ;
main()
{
   int n ; float x ;
   cout << "donnez un entier et un flottant\n" ;
   cin >> n >> x ;
   cout << "le produit de " << n << " par " << x << "\n'est :
" << n*x ;
}
```

Exercice 56

Énoncé

Écrire une fonction permettant d'échanger les contenus de 2 variables de type `int` fournies en argument :

a. en transmettant l'adresse des variables concernées (seule méthode utilisable en C) ;

b. en utilisant la transmission par référence.

Dans les deux cas, on écrira un petit programme d'essai (`main`) de la fonction.

Solutions

a. Avec la transmission des adresses des variables

```
#include <iostream>
using namespace std ;
main()
{ void echange (int *, int *) ;         // prototype de la
fonction echange
  int n=15, p=23 ;
  cout << "avant : " << n << " " << p << "\n" ;
  echange (&n, &p) ;
  cout << "après : " << n << " " << p << "\n" ;
}
void echange (int * a, int * b)
{ int c ;   // pour la permutation
  c = *a ;
  *a = *b ;
  *b = c ;
}
```

b. Avec une transmission par référence

```
#include <iostream>
using namespace std ;
main()
{ void echange (int &, int &) ;    // prototype de la fonction
echange
```

```
  int n=15, p=23 ;
  cout << "avant : " << n << " " << p << "\n" ;
  echange (n, p) ;                  // attention n et non &n, p
et non &p
  cout << "après : " << n << " " << p << "\n" ;
}
void echange (int & a, int & b)
{ int c ;                           // pour la permutation
  c = a ;
  a = b ;
  b = c ;
}
```

Exercice 57

Énoncé

Soit le modèle de structure suivant :

```
struct essai
{ int n ;
  float x ;
} ;
```

Écrire une fonction nommée `raz` permettant de remettre à zéro les 2 champs d'une structure de ce type transmise en argument :

a. par adresse ;

b. par référence.

Dans les deux cas, on écrira un petit programme d'essai de la fonction ; il affichera les valeurs d'une structure de ce type, après appel de ladite fonction.

Solutions

a. Avec une transmission d'adresse

```
#include <iostream>
using namespace std ;
```

```
struct essai
{ int n ;
  float x ;
} ;
void raz (struct essai * ads)
{ ads->n = 0 ;         // ou encore (*ads).n = 0 ;
  ads->x = 0.0 ;       // ou encore (*ads).x = 0.0 ;
}
main()
{ struct essai s ;
  raz (&s) ;
  cout << "valeurs après raz : " << s.n << " " << s.x ;
}
```

b. Avec une transmission par référence

```
#include <iostream>
using namespace std ;
struct essai
{ int n ;
  float x ;
} ;
void raz (struct essai & s)
{ s.n = 0 ;
  s.x = 0.0 ;
}
main()
{ struct essai s ;
  raz (s) ;                // notez bien s et non &s !
  cout << "valeurs après raz : " << s.n << " " << s.x ;
}
```

Exercice 58

Énoncé

Soient les déclarations (C++) suivantes :

```
int fct (int) ;          // fonction I
int fct (float) ;        // fonction II
void fct (int, float) ;  // fonction III
void fct (float, int) ;  // fonction IV
```

```
int n, p ;
float x, y ;
char c ;
double z ;
```

Les appels suivants sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions éventuellement mises en place ?

a. `fct (n) ;`

b. `fct (x) ;`

c. `fct (n, x) ;`

d. `fct (x, n) ;`

e. `fct (c) ;`

f. `fct (n, p) ;`

g. `fct (n, c) ;`

h. `fct (n, z) ;`

i. `fct (z, z) ;`

Solutions

Les cas **a**, **b**, **c** et **d** ne posent aucun problème. Il y a respectivement appel des fonctions I, II, III et IV, sans qu'aucune conversion d'argument ne soit nécessaire.

e. Appel de la fonction I, après conversion de la valeur de `c` en `int`.

f. Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : conserver `n`, convertir `p` en `float` et appeler la fonction III ou, au contraire, convertir `n en float`, conserver `p` et appeler la fonction IV.

g. Appel de la fonction III, après conversion de `c` en `float`.

h. Appel de la fonction III, après conversion (dégradante) de `z` en `float`.

i. Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : convertir le premier argument en `float` et le second en `int` et appeler la fonction III ou, au contraire, convertir le premier argument `en int` et le second `en float` et appeler la fonction IV. Notez que, dans les

deux cas, il s'agit de conversions dégradantes.

Exercice 59

Énoncé

Écrire plus simplement en C++ les instructions suivantes, en utilisant les opérateurs `new` et `delete` :

```
int * adi ;
double * add ;
   .....
adi = malloc (sizeof (int) ) ;
add = malloc (sizeof (double) * 100 ) ;
```

Solution

```
int * adi ;
double * add ;
   .....
adi = new int ;
add = new double [100] ;
```

On peut éventuellement tenir compte des possibilités de déclarations dynamiques offertes par C++ (c'est-à-dire que l'on peut introduire une déclaration à n'importe quel emplacement d'un programme), et écrire, par exemple :

```
int * adi = new int ;
double * add = new double [100] ;
```

Exercice 60

Énoncé

Écrire plus simplement en C++, en utilisant les spécificités de ce langage,

les instructions C suivantes :

```
double * adtab ;
int nval ;
   .....
printf ("combien de valeurs ? ") ;
scanf ("%d", &nval) ;
adtab = malloc (sizeof (double) * nval) ;
```

Solution

```
double * adtab ;
int nval ;
   .....
cout << "combien de valeurs ? " ;
cin >> nval ;
adtab = new double [nval] ;
```

Exercice 61

Énoncé

a. Transformer le programme suivant pour que la fonction `fct` devienne une fonction en ligne.

```
#include <iostream>
using namespace std ;
main()
{
    int fct (char, int) ;                  // déclaration
(prototype) de fct
   int n = 150, p ;
   char c = 's' ;
   p = fct ( c , n) ;
    cout << "fct (\'" << c << "\', " << n << ") vaut : " <<
p ;
}
int fct (char c, int n)            // définition de fct
{
   int res ;
   if (c == 'a')      res = n + c ;
```

```
    else if (c == 's') res = n - c ;
    else               res = n * c ;
    return res ;
}
```

b. Comment faudrait-il procéder si l'on souhaitait que la fonction `fct` soit compilée séparément ?

Solutions

a. Nous devons donc d'abord déclarer (et définir en même temps) la fonction `fct` comme une fonction en ligne. Le programme `main` s'écrit de la même manière, si ce n'est que la déclaration de `fct` n'y est plus nécessaire puisqu'elle apparaît auparavant.

```
#include <iostream>
using namespace std ;
inline int fct (char c, int n)
{   int res ;
    if (c == 'a')      res = n + c ;
    else if (c == 's') res = n - c ;
    else               res = n * c ;
    return res ;
}
main ()
{   int n = 150, p ;
    char c = 's' ;
    p = fct (c, n) ;
     cout << "fct (\'" << c << "\', " << n << ") vaut : " <<
p ;
}
```

b. Il s'agit en fait d'une question piège. En effet, la fonction `fct` étant en ligne, elle ne peut plus être compilée séparément. Il est cependant possible de la conserver dans un fichier d'extension `h` et d'incorporer simplement ce fichier par `#include` pour compiler le `main`. Cette démarche se rencontrera d'ailleurs fréquemment dans le cas de classes comportant des fonctions en ligne. Alors, dans un fichier d'extension `h`, on trouvera la déclaration de la classe en question, à l'intérieur de laquelle apparaîtront les « déclarations-définitions » des fonctions en ligne.

Exercice 62

Énoncé

a. Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std;
void f (int & n)
 { cout << "-- Appel f (int &)       " << "  n = " << n <<
endl ; }
void f (const int & n )
 { cout << "-- Appel f (const int &)" << "  n = " << n <<
endl ; }

int main()
{  int n = 10 ;
   const int p = 25 ;
   f(n) ;
   f(p) ;
   f(5) ;
   f(3*n+2) ;
   f(0.3*n) ;
   f(0.5) ;
}
```

b. Pourrait-on ajouter une troisième surdéfinition de **`f`** se présentant ainsi :

```
void f(int n)  { ..... }
```

Solutions

a.

Le premier appel `f(n)` conduit à choisir, parmi les deux fonctions `f` possibles, celle correspondant à une référence à une `lvalue`. En revanche, dans l'appel `f(p)`, `p` étant déclarée constante est une `lvalue` non modifiable et l'on choisit donc la fonction `f (const int&)`.

Dans les appels, `f(5)` et `f (3*n+2)`, les arguments ne sont pas des `lvalue` et c'est donc toujours la forme `f(const int &)` qui est choisie. Notez que le premier appel conduit à la création d'une copie temporaire de la valeur `5`.

Enfin, dans les deux derniers appels, on a affaire à des expressions de type `float` dont la valeur est convertie en `int` dans un emplacement temporaire fourni à `f(const int&)`.

En définitive, le programme affiche :

```
-- Appel f (int &)        n = 10
-- Appel f (const int &)  n = 25
-- Appel f (const int &)  n = 5
-- Appel f (const int &)  n = 32
-- Appel f (const int &)  n = 2
-- Appel f (const int &)  n = 0
```

b.

L'ajout de cette troisième surdéfinition de `f` créerait une ambiguïté puisque le mode de transmission d'un argument n'intervient pas dans le choix d'une fonction surdéfinie. On obtiendra une erreur de compilation.

Exercice 63

Énoncé

a. Quelle résultats fournit le programme suivant :

```
#include <iostream>
using namespace std;
int & h(int & n) { return n ; }
int main()
{  int qq = 25 ;
   cout << "-- En 1, h(qq) = " << h(qq) << endl ;
   int n = 10 ;
   cout << "-- En 2, n = " << n << endl ;
   h(n) = 20 ;
   cout << "-- En 3, n = " << n << endl ;
}
```

b. Que se passerait-il si l'on avait défini la fonction **h** de cette manière :

```
int h(int & n) { return n ; }
```

Solutions

a.

En 1, il s'agit d'un appel classique de `h`. Le fait que sa valeur de retour soit transmise par référence n'a pas d'incidence puisqu'on se contente de l'utiliser.

En revanche, l'appel :

```
h(n) = 20 ;
```

utilise `h(n)` comme une `lvalue`, ce qui est autorisé, car `h` renvoie son résultat par référence, résultat qui se trouve être précisément ici la référence à `n`. En définitive, cette instruction joue le même rôle que :

```
n = 20 ;
```

Finalement, le programme affiche :

```
-- En 1, h(qq) = 25
-- En 2, n = 10
-- En 3, n = 20
```

b.

Si l'on avait défini `h` de cette manière :

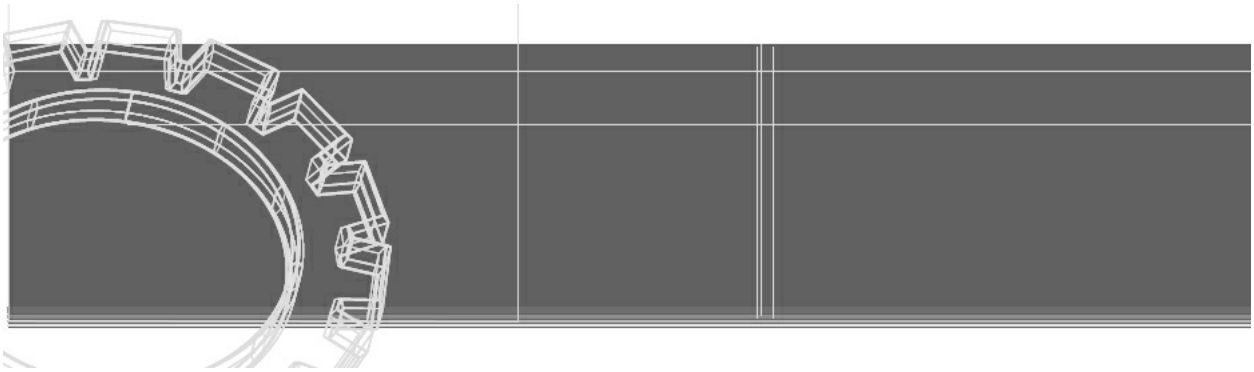
```
int h(int & n) { return n ; }
```

l'instruction :

```
 h(n) = 20 ;
```

provoquerait une erreur de compilation due à ce que `h(n)` n'est plus une `lvalue`.

Chapitre 7
Notions de classe, constructeur et destructeur



Rappels

Les possibilités de Programmation Orientée Objet de C++ reposent sur le concept de classe. Une classe est la généralisation de la notion de type défini par l'utilisateur, dans lequel se trouvent associées à la fois des données (on parle de « membres donnée ») et des fonctions (on parle de « fonctions membre » ou de méthodes). En P.O.O. pure, les données sont « encapsulées », ce qui signifie que leur accès ne peut se faire que par le biais des méthodes. C++ vous autorise à n'encapsuler qu'une partie seulement des données d'une classe.

Note

En toute rigueur, C++ vous permet également d'associer des fonctions

membre à des structures ou à des unions.

Déclaration et définition d'une classe

La **déclaration** d'une classe précise quels sont les membres (données ou fonctions) publics (c'est-à-dire accessibles à l'utilisateur de la classe) et quels sont les membres privés (inaccessibles à l'utilisateur de la classe). On utilise pour cela les mots-clés `public` et `private`, comme dans cet exemple dans lequel la classe `point` comporte deux membres donnée privés `x` et `y` et trois fonctions membre publiques `initialise`, `deplace` et `affiche` :

```
        /* ------------ Déclaration de la classe point --------
----- */
class point
{              /* déclaration des membres privés */
  private :
    int x ;
    int y ;
               /* déclaration des membres publics */
  public :
    void initialise (int, int) ;
    void deplace (int, int) ;
    void affiche () ;
} ;
```

La **définition** d'une classe consiste à fournir les définitions des fonctions membre. On indique alors le nom de la classe correspondante, à l'aide de l'opérateur de résolution de portée (`::`). Au sein de la définition même, les membres (privés ou publics — données ou fonctions) sont directement accessibles sans qu'il soit nécessaire de préciser le nom de la classe. Voici, par exemple, ce que pourrait être la définition de la fonction `initialise` de la classe précédente :

```
void point::initialise (int abs, int ord)
{
   x = abs ; y = ord ;
}
```

Ici, `x` et `y` représentent implicitement les membres `x` et `y` d'un objet de la classe `point`.

Remarques

1. Les mots-clés `public` et `private` peuvent apparaître à plusieurs reprises dans la déclaration d'une même classe. Une déclaration telle que la suivante est tout à fait envisageable :

```
class X
{     private :
       ...
      public :
       ...
      private :
       ...
      public :
       ...
} ;
```

2. Comme on le verra dans le chapitre consacré à l'héritage, il existe un troisième mot-clé, `protected`, qui n'intervient qu'en cas d'existence de classes dérivées.

Utilisation d'une classe

On déclare un « objet » d'un type classe donné en faisant précéder son nom de celui de la classe, comme dans l'instruction suivante qui déclare deux objets `a` et `b` de type `point` :

```
point a, b ;
```

On peut accéder à n'importe quel membre public (donnée ou fonction) d'une classe en utilisant l'opérateur ".". Par exemple :

```
a.initialise (5, 2) ;
```

appelle la fonction membre `initialise` de la classe à laquelle appartient l'objet `a`, c'est-à-dire, ici, la classe `point`.

Affectation entre objets

C++ autorise l'affectation d'un objet d'un type donné à un autre objet de même type. Dans ce cas, il y a (tout naturellement) recopie des valeurs des

champs de données (qu'ils soient publics ou privés). Toutefois, si, parmi ces champs, se trouvent des pointeurs, les emplacements pointés ne seront pas soumis à cette recopie. Si un tel effet est nécessaire (et il le sera souvent !), il ne pourra être obtenu qu'en « surdéfinissant » l'opérateur d'affectation pour la classe concernée (voyez le chapitre consacré à la surdéfinition d'opérateurs).

Constructeur et destructeur

Une fonction membre portant le même nom que sa classe se nomme un **constructeur**. Dès qu'une classe comporte un constructeur (au moins un), il n'est plus possible de déclarer un objet du type correspondant, sans fournir des valeurs pour les arguments requis par ce constructeur (sauf si ce dernier ne possède aucun argument). Le constructeur est appelé **après** l'allocation de l'espace mémoire destiné à l'objet.

Par définition, un constructeur ne renvoie pas de valeur (aucune indication de type, pas même `void`, ne doit figurer devant sa déclaration ou sa définition).

Une fonction membre portant le même nom que sa classe, précédé du symbole tilde (~), se nomme un **destructeur**. Le destructeur est appelé avant la libération de l'espace mémoire associé à l'objet. Par définition, un destructeur ne peut pas comporter d'arguments et il ne renvoie pas de valeur (aucune indication de type ne doit être prévue).

Membres donnée statiques

Un membre donnée déclaré avec l'attribut `static` est partagé par tous les objets de la même classe. Il existe même lorsque aucun objet de cette classe n'a été déclaré. Un membre donnée statique doit être initialisé explicitement, à l'extérieur de la classe (même s'il est privé), en utilisant l'opérateur de résolution de portée (`::`) pour spécifier sa classe. En général, son initialisation se fait dans la définition de la classe.

Exploitation d'une classe

En pratique, à l'utilisateur d'une classe (on dit souvent le « client »), on fournira :

- un fichier en-tête contenant la déclaration de la classe : l'utilisateur l'employant devra l'inclure dans tout programme faisant appel à la classe en question ;
- un module objet résultant de la compilation du fichier source contenant la définition de la classe, c'est-à-dire la définition de ses fonctions membre.

Exercice 64

Énoncé

Réaliser une classe `point` permettant de manipuler un point d'un plan. On prévoira :

- un constructeur recevant en arguments les coordonnées (`float`) d'un point ;
- une fonction membre `deplace` effectuant une translation définie par ses deux arguments (`float`) ;
- une fonction membre `affiche` se contentant d'afficher les coordonnées cartésiennes du point.

Les coordonnées du point seront des membres donnée privés.

On écrira séparément :

- un fichier source constituant la `déclaration` de la classe ;
- un fichier source correspondant à sa `définition`.

Écrire, par ailleurs, un petit programme d'essai (`main`) déclarant un point, l'affichant, le déplaçant et l'affichant à nouveau.

Solutions

a. Fichier source (nommé `point.h`) contenant la déclaration de la classe

```
/*       fichier POINT1.H         */
/* déclaration de la classe point */
```

```
class point
{
     float  x,  y  ;                                  //  coordonnées
(cartésiennes) du point
 public :
   point (float, float) ;          // constructeur
   void deplace (float, float) ;   // déplacement
   void affiche () ;               // affichage
} ;
```

b. Fichier source contenant la définition de la classe

```
        /* définition de la classe point */
#include "point1.h"
#include <iostream>
using namespace std ;
point::point (float abs, float ord)
{  x = abs ; y = ord ;
}
void point::deplace (float dx, float dy)
{  x = x + dx ; y = + dy ;
}
void point::affiche ()
{  cout << "Mes coordonnées cartésiennes sont " << x << " " <<
y << "\n" ;
}
```

Notez que, pour compiler ce fichier source, il est nécessaire d'inclure le fichier source, nommé ici `point1.h`, contenant la déclaration de la classe `point`.

c. Exemple d'utilisation de la classe

```
#include <iostream>
using namespace std ;
#include "point1.h"

main ()
{
     point  p  (1.25,  2.5)  ;   // construction  d'un  point  de
coordonnées 1.25 2.5
   p.affiche () ;          // affichage de ce point
   p.deplace (2.1, 3.4) ; // déplacement de ce point
   p.affiche () ;          // nouvel affichage
}
```

Bien entendu, pour pouvoir exécuter ce programme, il sera nécessaire d'introduire, lors de l'édition de liens, le module objet résultant de la

compilation du fichier source contenant la définition de la classe `point`.

Notez que, généralement, le fichier `point1.h` contiendra des directives conditionnelles de compilation, afin d'éviter les risques d'inclusion multiple. Par exemple, on pourra procéder ainsi :

```
#ifndef POINT1_H
#define POINT1_H
  .....
  déclaration de la classe.....
  .....
#endif
```

Exercice 65

Énoncé

Réaliser une classe `point`, analogue à la précédente, mais ne comportant pas de fonction `affiche`. Pour respecter le principe d'encapsulation des données, prévoir deux fonctions membre publiques (nommées `abscisse` et `ordonnee`) fournissant en retour l'abscisse et l'ordonnée d'un point. Adapter le petit programme d'essai précédent pour qu'il fonctionne avec cette nouvelle classe.

Solution

Il suffit d'introduire deux nouvelles fonctions membre `abscisse` et `ordonnee` et de supprimer la fonction `affiche`. La nouvelle déclaration de la classe est alors :

```
        /*       fichier POINT2.H         */
        /* déclaration de la classe point */
class point
{
     float  x,  y  ;                              //  coordonnées
(cartésiennes) du point
 public :
   point (float, float) ;          // constructeur
   void deplace (float, float) ;  // déplacement
```

```
   float abscisse () ;            // abscisse du point
   float ordonnee () ;            // ordonnée du point
} ;
```

Voici sa nouvelle définition :

```
        /* définition de la classe point */
#include "point2.h"
#include <iostream>
using namespace std ;
point::point (float abs, float ord)
{  x = abs ; y = ord ;
}
void point::deplace (float dx, float dy)
{  x = x + dx ; y = + dy ;
}
float point::abscisse ()
{ return x ;
}
float point::ordonnee ()
{ return y ;
}
```

Et le nouveau programme d'essai :

```
   /* exemple d'utilisation de la classe point */
#include "point2.h"
#include <iostream>
using namespace std ;
main ()
{
   point p (1.25, 2.5) ;                                    //
construction
                                                            //
affichage
   cout << "Coordonnées cartésiennes : " << p.abscisse () << "
"
                                          << p.ordonnee () <<
"\n" ;
   p.deplace (2.1, 3.4) ;                                   //
déplacement
                                                            //
affichage
   cout << "Coordonnées cartésiennes : " << p.abscisse () << "
"
                                          << p.ordonnee () <<
"\n" ;
}
```

Discussion

Cet exemple montre qu'il est toujours possible de respecter le principe d'encapsulation en introduisant ce que l'on nomme des « fonctions d'accès ». Il s'agit de fonctions membre destinées à accéder (aussi bien en consultation – comme ici – qu'en modification) aux membres privés. L'intérêt de leur emploi (par rapport à un accès direct aux données qu'il faudrait alors rendre publiques) réside dans la souplesse de modification de l'implémentation de la classe qui en découle. Ainsi, ici, il est tout à fait possible de modifier la manière dont un point est représenté (par exemple, en utilisant ses coordonnées polaires plutôt que ses coordonnées cartésiennes), sans que l'utilisateur de la classe n'ait à se soucier de cet aspect. C'est d'ailleurs ce que vous montrera l'exercice **67** ci-après.

Exercice 66

Énoncé

Ajouter à la classe précédente (comportant un constructeur et trois fonctions membre `deplace`, `abscisse` et `ordonnee`) de nouvelles fonctions membre :

- `homothetie` qui effectue une homothétie dont le rapport est fourni en argument ;
- `rotation` qui effectue une rotation dont l'angle est fourni en argument ;
- `rho` et `theta` qui fournissent en retour les **coordonnées polaires** du point.

Solution

La déclaration de la nouvelle classe `point` découle directement de l'énoncé :

```
/*       fichier POINT3.H         */
/* déclaration de la classe point */
```

```
class point
{   float  x,  y  ;                                  //  coordonnées
(cartésiennes) du point
 public :
   point (float, float) ;          // constructeur
   void deplace (float, float) ;   // déplacement
   void homothetie (float) ;       // homothétie
   void rotation (float) ;         // rotation
   float abscisse () ;             // abscisse du point
   float ordonnee () ;             // ordonnée du point
   float rho () ;                  // rayon vecteur
   float theta () ;                // angle
} ;
```

Sa définition mérite quelques remarques. En effet, si `homothetie` ne présente aucune difficulté, la fonction membre `rotation` nécessite quant à elle une transformation intermédiaire des coordonnées cartésiennes du point en coordonnées polaires. De même, la fonction membre `rho` doit calculer le rayon vecteur d'un point dont on connaît les coordonnées cartésiennes tandis que la fonction membre `theta` doit calculer l'angle d'un point dont on connaît les coordonnées cartésiennes.

Le calcul de rayon vecteur étant simple, nous l'avons laissé figurer dans les deux fonctions concernées (`rotation` et `rho`). En revanche, le calcul d'angle a été réalisé par ce que nous nommons une « fonction de service », c'est-à-dire une fonction qui n'a d'intérêt que dans la définition de la classe elle-même. Ici, il s'agit d'une fonction indépendante mais, bien entendu, on peut prévoir des fonctions de service sous forme de fonctions membre (elles seront alors généralement privées).

Voici finalement la définition de notre classe `point`.

```
            /****************    déclarations    de    service
******************/
#include "point3.h"
#include <cmath>                        // pour sqrt et atan
#include <iostream>
using namespace std ;
const float pi = 3.141592653 ;          // valeur de pi
float angle (float, float) ;              // fonction de service
(non membre)
        /***************   définition   des   fonctions   membre
**********/
point::point (float abs, float ord)
{  x = abs ; y = ord ;
```

```
}
void point::deplace (float dx, float dy)
{  x += dx ; y += dy ;
}
void point::homothetie (float hm)
{  x *= hm ; y *= hm ;
}
void point::rotation (float th)
{  float r = sqrt (x*x + y*y) ;       // passage en
   float t = angle (x, y) ;           // coordonnées polaires
   t += th ;                          // rotation th
   x = r * cos (t) ;                  // retour en
    y = r * sin (t) ;                          // coordonnées
cartésiennes
}
float point::abscisse ()
{ return x ;
}
float point::ordonnee ()
{ return y ;
}
float point::rho ()
{ return sqrt (x*x + y*y) ;
}
float point::theta ()
{ return angle (x, y) ;
}
            /********** définition des fonctions de service
***********/
/*   fonction   de   calcul   de   l'angle   correspondant   aux
coordonnées     */
/*            cartésiennes fournies en argument
      */
/*  On  choisit  une  détermination  entre  -pi  et  +pi  (0  si
x=0)        */
float angle (float x, float y)
{  float a = x ? atan (y/x) : 0 ;
   if (x<0) if (y>=0) return a + pi ;
                 else return a - pi ;
   return a ;
}
```

Exercice 67

Énoncé

Modifier la classe `point` précédente, de manière que les données (privées) soient maintenant les coordonnées polaires d'un point, et non plus ses coordonnées cartésiennes. On évitera de modifier la déclaration des membres publics, de sorte que l'interface de la classe (ce qui est visible pour l'utilisateur) ne change pas.

Solution

La déclaration de la nouvelle classe découle directement de l'énoncé :

```
        /*       fichier POINT4.H  :  déclaration de la classe
point */
class point
{  float r, t ;                    // coordonnées (polaires) du
point
 public :
   point (float, float) ;          // constructeur
   void deplace (float, float) ;   // déplacement
   void homothetie (float) ;       // homothétie
   void rotation (float) ;         // rotation
   float abscisse () ;             // abscisse du point
   float ordonnee () ;             // ordonnée du point
   float rho () ;                  // rayon vecteur
   float theta () ;                // angle
} ;
```

En ce qui concerne sa définition, il est maintenant nécessaire de remarquer que :

- le constructeur reçoit toujours en argument les coordonnées cartésiennes d'un point ; il doit donc opérer les transformations appropriées ;
- la fonction `deplace` reçoit un déplacement exprimé en coordonnées cartésiennes ; il faut donc tout d'abord déterminer les coordonnées cartésiennes du point après déplacement, avant de repasser en coordonnées polaires.

En revanche, les fonctions `homothetie` et `rotation` s'expriment très simplement.

Voici la définition de notre nouvelle classe (nous avons fait appel à la même

fonction de service `angle` que dans l'exercice précédent) :

```
#include "point4.h"
#include <cmath>                          // pour cos, sin, sqrt
et atan
#include <iostream>
using namespace std ;
const int pi = 3.141592635 ;          // valeur de pi

  /********** définition des fonctions de service ***********/
/*   fonction   de   calcul   de   l'angle   correspondant   aux
coordonnées     */
/*            cartésiennes fournies en argument
      */
/*  On  choisit  une  détermination  entre  -pi  et  +pi  (0  si
x=0)        */
float angle (float x, float y)
{  float a = x ? atan (y/x) : 0 ;
   if (x<0) if (y>=0) return a + pi ;
     else return a - pi ;
   return a ;
}
        /**********    définition    des    fonctions    membre
*****************/
point::point (float abs, float ord)
{  r = sqrt (abs*abs + ord*ord) ;
   t = atan (ord/abs) ;
}
void point::deplace (float dx, float dy)
{  float x = r * cos (t) + dx ;   // nouvelle abscisse
   float y = r * sin (t) + dy ;   // nouvelle ordonnée
   r = sqrt (x*x + y*y) ;
   t = angle (x, y) ;
}
void point::homothetie (float hm)
{  r *= hm ;
}
void point::rotation (float th)
{  t += th ;
}
float point::abscisse ()
{ return r * cos (t) ;
}
float point::ordonnee ()
{ return r * sin (t) ;
}
float point::rho ()
{ return r ;
```

```
}
float point::theta ()
{ return t ;
}
```

Exercice 68

Énoncé

Soit la classe `point` créée dans l'exercice 62, dont la déclaration était la suivante :

```
class point
{  float x, y ;
  public :
   point (float, float) ;
   void deplace (float, float) ;
   void affiche () ;
}
```

Adapter cette classe, de manière que la fonction membre `affiche` fournisse, en plus des coordonnées du point, le nombre d'objets de type `point`.

Solution

Il faut donc définir un compteur du nombre d'objets existant à un moment donné. Ce compteur doit être incrémenté à chaque création d'un nouvel objet, donc par le constructeur `point`. De même, il doit être décrémenté à chaque destruction d'un objet, donc par le destructeur de la classe `point` ; il faudra donc ajouter ici une fonction membre nommée `~point`.

Quant au compteur proprement dit, nous pourrions certes en faire une variable globale, définie par exemple en même temps que la classe ; cette démarche présente toutefois des risques d'effets de bord (modification accidentelle de la valeur de cette variable, depuis n'importe quel programme utilisateur). Il est plus judicieux d'en faire un membre privé statique.

Voici la nouvelle déclaration de notre classe `point` :

```
        /*       fichier POINT5.H         */
        /* déclaration de la classe point */
class point
{
    static nb_pts ;                    // compteur du nombre
d'objets créés
     float x, y ;                              // coordonnées
(cartésiennes) du point
 public :
   point (float, float) ;          // constructeur
   ~point () ;                     // destructeur
   void deplace (float, float) ;   // déplacement
   void affiche () ;               // affichage
} ;
```

Et voici sa nouvelle définition (notez l'initialisation du membre statique) :

```
#include "point5.h"
#include <iostream>
using namespace std ;
int point::nb_pts = 0 ;                    // initialisation
obligatoire
                                          // statique nb_pts
point::point (float abs, float ord)       // constructeur
   x = abs ; y = ord ;
    nb_pts++ ;                               // actualisation nb
points
}
point::~point ()                          // destructeur
{  nb_pts-- ;                               // actualisation nb
points
}
void point::deplace (float dx, float dy)
{  x = x + dx ; y = + dy ;
}
void point::affiche ()
{ cout << "Je suis un point parmi " << nb_pts
  << " de coordonnées "<< x << " " << y << "\n" ;
}
```

Remarque

Il pourrait être judicieux de munir notre classe `point` d'une fonction membre fournissant le nombre d'objets de type `point` existant à un moment donné. C'est ce que nous vous proposerons dans un exercice du

prochain chapitre.

Exercice 69

Énoncé

Réaliser une classe nommée `set_char` permettant de manipuler des ensembles de caractères. On devra pouvoir réaliser sur un tel ensemble les opérations classiques suivantes : lui ajouter un nouvel élément, connaître son « cardinal » (nombre d'éléments), savoir si un caractère donné lui appartient.

Ici, on n'effectuera aucune allocation dynamique d'emplacements mémoire. Il faudra donc prévoir, en membre donnée, un tableau de taille fixe.

Écrire, en outre, un programme (`main`) utilisant la classe `set_char` pour déterminer le nombre de caractères différents contenus dans un mot lu en donnée.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, et qu'il ne faut pas chercher à utiliser ici.

Solution

Compte tenu des contraintes imposées par l'énoncé (pas de gestion dynamique), une solution consiste à prévoir un tableau dans lequel un élément de rang `i` précise si le caractère de code `i` appartient ou non à l'ensemble. Notez qu'il est nécessaire que `i` soit positif ou nul ; on travaillera donc toujours sur des caractères non signés. La taille du tableau doit être égale au nombre de caractères qu'il est possible de représenter dans une implémentation donnée (généralement 256).

Le reste de la déclaration de la classe découle de l'énoncé.

```
/*          fichier SETCHAR1.H          */
```

```
        /* déclaration de la classe set_char */
const int N_CAR_MAX=256 ;   // on pourrait utiliser UCHAR_MAX
défini
                            // dans <climits>

class set_char
{
   unsigned char ens [N_CAR_MAX] ;
                                    //  tableau  des  indicateurs
(présent/absent)
                       // pour chacun des caractères possibles
 public :
   set_char () ;                     // constructeur
   void ajoute (unsigned char) ;     // ajout d'un élément
    int  appartient  (unsigned  char)  ;   //  appartenance  d'un
élément
      int cardinal () ;              // cardinal de l'ensemble
} ;
```

La définition de la classe en découle assez naturellement :

```
   /* définition de la classe set_char */
#include "setchar1.h"
set_char::set_char ()
{  int i ;
   for (i=0 ; i<N_CAR_MAX ; i++) ens[i] = 0 ;
}

void set_char::ajoute (unsigned char c)
{   ens[c] = 1 ;
}

int set_char::appartient (unsigned char c)
{   return ens[c] ;
}

int set_char::cardinal ()
{   int i, n ;
    for (i=0, n=0 ; i<N_CAR_MAX ; i++) if (ens[i]) n++ ;
    return n ;
}
```

Il en va de même pour le programme d'utilisation :

```
   /* utilisation de la classe set_char */
#include <cstring>
#include "setchar1.h"
#include <iostream>
using namespace std ;
```

```
main()
{  set_char ens ;
   char mot [81] ;
   cout << "donnez un mot " ;
   cin >> mot ;
   int i ;
   for (i=0 ; i<strlen(mot) ; i++) ens.ajoute (mot[i]) ;
    cout << "il contient " << ens.cardinal () << " caractères
différents" ;
     if (ens.appartient('e')) cout << "le caractère e est
présent\n" ;
                        else cout << "le caractère e n'est pas
présent\n" ;
}
```

Remarque

Si l'on avait déclaré de type `char` les arguments de `ajoute` et `appartient`, on aurait alors pu aboutir soit au type `unsigned char`, soit au type `signed char`, selon l'environnement utilisé. Dans le dernier cas, on aurait couru le risque de transmettre à l'une des fonctions membre citées une valeur négative, et partant d'accéder à l'extérieur du tableau `ens`.

Discussion

Le tableau `ens [N_CHAR_MAX]` occupe un octet par caractère ; chacun de ces octets ne prend que l'une des valeurs `0` ou `1` ; on pourrait économiser de l'espace mémoire en prévoyant seulement 1 bit par caractère. Les fonctions membre y perdraient toutefois en simplicité, ainsi qu'en vitesse.

Bien entendu, beaucoup d'autres implémentations sont possibles ; c'est ainsi, par exemple, que l'on pourrait fournir au constructeur un nombre maximal d'éléments, et allouer dynamiquement l'emplacement mémoire correspondant ; toutefois, là encore, on perdrait le bénéfice de la correspondance immédiate entre un caractère et la position de son indicateur. Notez toutefois que ce sera la seule possibilité réaliste lorsqu'il s'agira de représenter des ensembles dans lesquels le nombre maximal d'éléments sera très grand.

Exercice 70

Énoncé

Modifier la classe `set_char` précédente, de manière à disposer de ce que l'on nomme un « itérateur » sur les différents éléments de l'ensemble. Il s'agit d'un mécanisme permettant d'accéder séquentiellement aux différents éléments. On prévoira trois nouvelles fonctions membre : `init`, qui initialise le processus d'exploration ; `prochain`, qui fournit la valeur de l'élément suivant lorsqu'il existe et `existe`, qui précise s'il existe encore un élément non exploré.

On complétera alors le programme d'utilisation précédent, de manière qu'il affiche les différents caractères contenus dans le mot fourni en donnée.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, et qu'il ne faut pas chercher à utiliser ici.

Solution

Compte tenu de l'implémentation de notre classe, la gestion du mécanisme d'itération nécessite l'emploi d'un indice (que nous nommerons `courant`) sur un élément du tableau `ens`. Nous conviendrons que `courant` désigne le premier élément de `ens` non encore traité dans l'itération, c'est-à-dire non encore renvoyé par la fonction membre `suivant` (nous aurions pu adopter la convention contraire, à savoir que `courant` désigne le dernier élément traité).

En outre, pour faciliter la reconnaissance de la fin de l'itération, nous utiliserons un membre donnée supplémentaire (`fin`) valant `0` dans les cas usuels, et `1` lorsqu'aucun élément ne sera disponible (pour `suivant`).

Le rôle de la fonction `init` sera donc de faire pointer `courant` sur la première valeur non nulle de `ens` s'il en existe une ; dans le cas contraire, `fin` sera placé à `1`.

La fonction `suivant` fournira en retour l'élément pointé par `courant` lorsqu'il existe (`fin` non nul) ou la valeur `0` dans le cas contraire (il s'agit là d'une

convention destinée à protéger l'utilisateur ayant appelé cette fonction, alors qu'aucun élément n'était plus disponible). Dans le premier cas, `suivant` recherchera le prochain élément de l'ensemble (en modifiant la valeur de `fin` lorsqu'un tel élément n'existe pas). Notez bien qu'ici la fonction `suivant` doit renvoyer non pas le prochain élément, mais l'élément courant.

Enfin, la fonction `existe` se contentera de renvoyer la valeur de `fin` puisque cette dernière indique l'existence ou l'inexistence d'un élément `courant`.

Voici la nouvelle définition de la classe `set_char` :

```
          /*         fichier SETCHAR2.H        */
          /* déclaration de la classe set_char */
  const int N_CAR_MAX=256      // on pourrait utiliser UCHAR_MAX
  défini
                               // dans <climits>

  class set_char
  {
     unsigned char ens [N_CAR_MAX] ;
        // tableau des indicateurs (présent/absent)
        // pour chacun des caractères possibles
     int courant ;  // position courante dans le tableau ens
     int fin ;      // indique si fin atteinte
   public :
     set_char () ;                       // constructeur
     void ajoute (unsigned char) ;       // ajout d'un élément
      int appartient (unsigned char) ;   // appartenance d'un
  élément
     int cardinal () ;                   // cardinal de l'ensemble
      void init () ;                             // initialisation
  itération
     unsigned char suivant () ;          // caractère suivant
     int existe () ;
  } ;
```

Voici la définition des trois nouvelles fonctions membre `init`, `suivant` et `existe` :

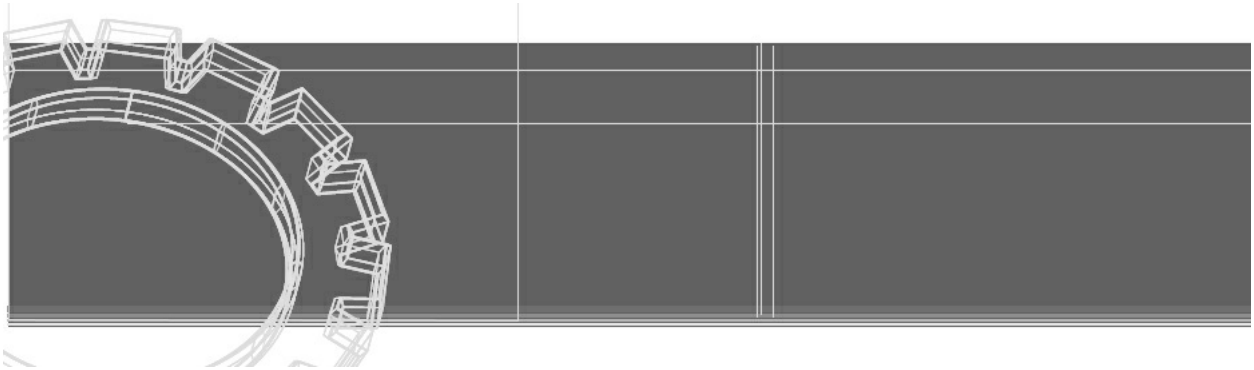
```
  void set_char::init ()
  {  courant=0 ; fin = 0 ;
     while ( (++courant<N_CAR_MAX) && (!ens[courant]) ) ;
     // si la fin de ens est atteinte, courant vaut N_CAR_MAX
     if (courant>=N_CAR_MAX) fin = 1 ;
  }

  unsigned char set_char::suivant ()
```

```
{   if (fin) return 0 ;  // au cas où on serait déjà en fin de
ens
    unsigned char c = courant ;   // conservation du caractère
courant
       // et recherche du suivant s'il existe
    while ( (++courant<N_CAR_MAX) && (!ens[courant]) )  ;
           // si la fin de ens est atteinte, courant vaut
N_CAR_MAX
     if (courant>=N_CAR_MAX) fin = 1 ;   // s'il n'y a plus de
caractère
    return c ;
}
int set_char::existe ()
{   return (!fin) ;
}
```

Voici enfin l'adaptation du programme d'utilisation :

```
#include <cstring>
#include "setchar2.h"
#include <iostream>
using namespace std ;
main()
{  set_char ens ;
   char mot [81] ;
   cout << "donnez un mot " ;
   cin >> mot ;
   int i ;
   for (i=0 ; i<strlen(mot) ; i++) ens.ajoute (mot[i]) ;
    cout << "il contient " << ens.cardinal () << " caractères
différents"
        << " qui sont :\n" ;
     ens.init()  ;  // init itération sur les caractères de
l'ensemble
   while (ens.existe())
     cout << ens.suivant () ;
}
```

Chapitre 8
Propriétés des fonctions membre



Rappels

Surdéfinition des fonctions membre et arguments par défaut

Il s'agit simplement de la généralisation aux fonctions membre des possibilités déjà offertes par C++ pour les fonctions « ordinaires ».

Fonctions membre en ligne

Il s'agit également de la généralisation aux fonctions membre d'une possibilité offerte pour les fonctions ordinaires, avec une petite nuance concernant sa mise en œuvre ; pour rendre « en ligne » une fonction membre, on peut :

- soit fournir directement la définition de la fonction dans la déclaration même de la classe ; dans ce cas, le qualificatif `inline` n'a pas à être

utilisé, comme dans cet exemple :

```
class truc
{   ...
    int fctenlig (int, float)
      {    définition de fctenlig
      }
    .....
} ;
```

- soit procéder comme pour une fonction ordinaire, en fournissant une définition en dehors de la déclaration de la classe ; dans ce cas, le qualificatif `inline` doit apparaître, à la fois devant la déclaration et devant l'en-tête.

Cas des objets transmis en arguments d'une fonction membre

Une fonction membre reçoit implicitement l'adresse de l'objet l'ayant appelé. Mais, en outre, il est toujours possible de lui transmettre explicitement un argument (ou plusieurs) du type de sa classe, ou même du type d'une autre classe. Dans le premier cas, la fonction membre aura accès aux membres privés de l'argument en question (car, en C++, l'unité d'encapsulation est la classe elle-même et non l'objet). En revanche, dans le second cas, la fonction membre n'aura accès qu'aux membres publics de l'argument.

Un tel argument peut être transmis classiquement par valeur, par adresse ou par référence. Avec la transmission par valeur, il y a recopie des valeurs des membres donnée dans un emplacement local à la fonction appelée. Des problèmes peuvent surgir dès lors que l'objet transmis en argument contient des pointeurs sur des parties dynamiques. Ils seront réglés par l'emploi d'un « constructeur par recopie » (voir chapitre suivant).

Remarque

Bien que ce soit d'un usage plus limité, une fonction ordinaire peut également recevoir un argument de type classe. Bien entendu, elle n'aura alors accès qu'aux membres publics de cet argument (à moins d'avoir été déclarée fonction amie, comme on le verra dans le chapitre correspondant).

Cas des fonctions membre fournissant un objet en retour

Une fonction membre peut fournir comme valeur de retour un objet du type de sa classe ou d'un autre type classe (dans ce dernier cas, elle n'accédera bien sûr qu'aux membres publics de l'objet en question). La transmission peut, là encore, se faire par valeur, par adresse ou par référence.

La transmission par valeur implique une recopie qui pose donc les mêmes problèmes que ceux évoqués ci-dessus pour les objets comportant des pointeurs sur des parties dynamiques. Quant aux transmissions par adresse ou par référence, elles doivent être utilisées avec beaucoup de précautions, dans la mesure où, dans ce cas, on renvoie (généralement) l'adresse d'un objet alloué automatiquement, c'est-à-dire dont la durée de vie coïncide avec celle de la fonction.

Autoréférence : le mot-clé this

Au sein d'une fonction membre, `this` représente un pointeur sur l'objet ayant appelé ladite fonction membre.

Fonctions membre statiques

Lorsqu'une fonction membre a une action indépendante d'un quelconque objet de sa classe, on peut la déclarer avec l'attribut `static`. Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (`::`).

Fonctions membre constantes

On peut déclarer des objets constants (à l'aide du qualificatif `const`). Dans ce cas, seules les fonctions membre déclarées (et définies) avec ce même qualificatif (exemple de déclaration : `void affiche () const`) peuvent recevoir (implicitement ou explicitement) en argument un objet constant.

Exercice 71

Énoncé

On souhaite réaliser une classe `vecteur3d` permettant de manipuler des vecteurs à trois composantes. On prévoit que sa déclaration se présente ainsi :

```
class vecteur3d
{    float x, y, z ;          // pour les 3 composantes
(cartésiennes)
    .....
} ;
```

On souhaite pouvoir déclarer un vecteur, soit en fournissant explicitement ses trois composantes, soit en en fournissant aucune, auquel cas le vecteur créé possédera trois composantes nulles. Écrire le ou les constructeurs correspondants :

a. en utilisant des fonctions membre surdéfinies ;

b. en utilisant une seule fonction membre ;

c. en utilisant une seule fonction en ligne.

Solutions

Il s'agit de simples applications des possibilités de surdéfinition, d'arguments par défaut et d'écriture en ligne des fonctions membre.

a.

```
          /* déclaration de la classe vecteur3d */
class vecteur3d
{   float x, y, z ;
   public :
     vecteur3d () ;                      // constructeur sans
arguments
      vecteur3d (float, float, float) ;   // constructeur 3
composantes
    .....
} ;

    /* définition des constructeurs de la classe vecteur3d */
vecteur3d::vecteur3d ()
{ x = 0 ; y = 0 ; z = 0 ;
}
```

```
vecteur3d::vecteur3d (float c1, float c2, float c3)
{ x = c1 ; y = c2 ; z = c3 ;
}
```

b.

```
          /* déclaration de la classe vecteur3d */
class vecteur3d
{   float x, y, z ;
   public :
        vecteur3d  (float=0.0,  float=0.0,  float=0.0)  ;  //
constructeur (unique)
    .....
} ;
    /* définition du constructeur de la classe vecteur3d */
vecteur3d::vecteur3d (float c1, float c2, float c3)
{ x = c1 ; y = c2 ; z = c3 ;
}
```

On notera toutefois qu'avec ce constructeur il est possible de déclarer un point en fournissant non seulement zéro ou trois composantes, mais éventuellement seulement une ou deux. Cette solution n'est donc pas rigoureusement équivalente à la précédente.

c.

```
          /* déclaration de la classe vecteur3d */
class vecteur3d
{   float x, y, z ;
   public :
      // constructeur unique "en ligne"
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
    { x = c1 ; y = c2 ; z = c3 ;
    }
   .....
} ;
```

Ici, il n'y a plus aucune définition de constructeur, puisque ce dernier est en ligne.

Exercice 72

Énoncé

Soit une classe `vecteur3d` définie comme suit :

```
class vecteur3d
{   float x, y, z ;
   public :
    vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
      { x = c1 ; y = c2 ; z = c3 ;
      }
     .....
} ;
```

Introduire une fonction membre nommée `coincide` permettant de savoir si deux vecteurs ont les mêmes composantes :

a. en utilisant une transmission par valeur ;

b. en utilisant une transmission par adresse ;

c. en utilisant une transmission par référence.

Si `v1` et `v2` désignent 2 vecteurs de type `vecteur3d`, comment s'écrit le test de coïncidence de ces 2 vecteurs, dans chacun des 3 cas considérés ?

Solutions

La fonction `coincide` est membre de la classe `vecteur3d` ; elle recevra donc implicitement l'adresse du vecteur l'ayant appelé. Elle ne possédera donc qu'un seul argument, lui-même de type `vecteur3d`. Nous supposerons qu'elle fournit une valeur de retour de type `int` (`1` pour la coïncidence, `0` dans le cas contraire).

a. La déclaration de `coincide` pourra se présenter ainsi :

```
int coincide (vecteur3d) ;
```

Voici ce que pourrait être sa définition :

```
int vecteur3d::coincide (vecteur3d v)
{   if ( (v.x == x) && (v.y == y) && (v.z == z) ) return 1 ;
                                                  else return 0 ;
}
```

b. La déclaration de `coincide` devient :

```
int coincide (vecteur3d *) ;
```

Et sa nouvelle définition pourrait être :

```
int vecteur3d::coincide (vecteur3d * adv)
{   if ( (adv->x == x) && (adv->y == y) && (adv->z == z)
                return 1 ;
            else return 0 ;
}
```

Remarque

En utilisant `this`, la définition de `coincide` pourrait faire moins de distinction entre ses deux arguments (l'un implicite, l'autre explicite) :

```
int vecteur3d::coincide (vecteur3d * adv)
{  if ( (adv->x == this->x) && (adv->y == this->y) && (adv-
>z == this->z) )
                 return 1 ;
             else return 0 ;
}
```

---

c. La déclaration de `coincide` devient :

```
int coincide (vecteur3d &) ;
```

Et sa nouvelle définition est :

```
int vecteur3d::coincide (vecteur3d & v)
{   if ( (v.x == x) && (v.y == y) && (v.z == z) ) return 1 ;
                                                  else return 0 ;
}
```

Notez que le corps de la fonction est resté le même qu'en **a**.

Voici les trois appels de `coincide` correspondant respectivement aux trois définitions précédentes :

a. `v1.coincide (v2)` ; ou `v2.coincide (v1)` ;

b. `v1.coincide (&v2)` ou `v2.coincide (&v1)` ;

c. `v1.coincide (v2)` ou `v2.coincide (v1)` ;

Discussion

La surdéfinition d'opérateur offrira une mise en œuvre plus agréable de ce test de coïncidence de deux vecteurs. C'est ainsi qu'il sera possible de surdéfinir l'opérateur de comparaison `==` (pour la classe `vecteur3d`) et, partant, d'exprimer ce test sous la simple forme `v1 == v2`.

Exercice 73

Énoncé

Soit une classe `vecteur3d` définie comme suit :

```
class vecteur3d
{   float x, y, z ;
   public :
    vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
      { x = c1 ; y = c2 ; z = c3 ;
      }
     .....
} ;
```

Introduire, dans cette classe, une fonction membre nommée `normax` permettant d'obtenir, parmi deux vecteurs, celui qui a la plus grande norme. On prévoira trois situations :

a. le résultat est renvoyé par valeur ;

b. le résultat est renvoyé par référence, l'argument (explicite) étant également transmis par référence ;

c. le résultat est renvoyé par adresse, l'argument (explicite) étant également transmis par adresse.

Solutions

a. La seule difficulté réside dans la manière de renvoyer la valeur de l'objet ayant appelé une fonction membre, à savoir `*this`. Voici la définition de la fonction `normax` (la déclaration en découle immédiatement) :

```
vecteur3d vecteur3d::normax (vecteur3d v)
{  float norm1 = x*x + y*y + z*z ;
```

```
        float norm2 = v.x*v.x + v.y*v.y + v.z*v.z ;
        if (norm1>norm2) return *this ;
                    else return v ;
    }
```

Voici un exemple d'utilisation (on suppose que `v1`, `v2` et `w` sont de type `vecteur3d`*) :*

```
w = v1.normax (v2) ; /* on obtient dans w celui des deux
vecteurs v1 et v2 */
                     /*  ayant la plus grande norme
            */
```

Notez bien que l'affectation ne pose aucun problème ici, puisque notre classe ne comporte aucun pointeur sur des parties dynamiques.

b. Aucun nouveau problème ne se pose. Il suffit de modifier ainsi l'en-tête de notre fonction, sans en modifier le corps :

```
vecteur3d & vecteur3d::normax (vecteur3d & v)
```

La fonction `normax` s'utilise comme précédemment.

c. Il faut, cette fois, adapter en conséquence l'en-tête et le corps de la fonction :

```
vecteur3d *  vecteur3d::normax (vecteur3d * adv)
{
   float norm1 = x * x + y * y + z * z ;
    float norm2 = adv->x * adv->x + adv->y * adv->y + adv->z *
adv->z ;
   if (norm1>norm2) return this ;
               else return  adv ;
}
```

Ici, l'utilisation de la fonction nécessite quelques précautions. En voici un exemple (`v1`, `v2` et `w` sont toujours de type `vecteur3d`) :

```
w = *(v1.normax(&v2))  ;
```

Discussion

En ce qui concerne la transmission de l'unique argument explicite de `normax`, il faut noter qu'il est impossible de la prévoir par valeur, dès lors que `normax` doit restituer son résultat par adresse ou par référence. En effet, dans ce cas, on obtiendrait en retour l'adresse ou la référence d'un vecteur alloué automatiquement au sein de la fonction. Notez qu'un tel problème ne se pose

pas pour l'argument implicite (`this`), car il correspond toujours à l'adresse d'un vecteur (transmis automatiquement par référence), et non à une valeur.

Par ailleurs, on ne perdra pas de vue qu'il est rare qu'une fonction puisse renvoyer l'adresse ou la référence d'un objet. Ici, la chose n'est possible que parce que ce résultat n'a pas été créé dynamiquement dans la fonction.

Exercice 74

Énoncé

Réaliser une classe `vecteur3d` permettant de manipuler des vecteurs à 3 composantes (de type `float`). On y prévoira :

- un constructeur, avec des valeurs par défaut (`0`),
- une fonction d'affichage des 3 composantes du vecteur, sous la forme :

```
< composante1, composante2, composante3 >
```

- une fonction permettant d'obtenir la somme de 2 vecteurs ;
- une fonction permettant d'obtenir le produit scalaire de 2 vecteurs.

On choisira les modes de transmission les mieux appropriés. On écrira un petit programme utilisant la classe ainsi réalisée.

Solution

La fonction membre calculant la somme de deux vecteurs (nous la nommerons `somme`) reçoit implicitement (par référence) un argument de type `vecteur3d`. Elle comportera donc un seul argument, lui aussi de type `vecteur3d`. On peut, a priori, le transmettre par adresse, par valeur ou par référence. En fait, la transmission par adresse, en C++, n'a plus guère de raison d'être, dans la mesure où la transmission par référence fait la même chose, moyennant une écriture plus agréable.

Le choix doit donc se faire entre transmission par valeur ou par référence.

Lorsqu'il s'agit de transmettre un objet (comportant plusieurs membres donnée), la transmission par référence est plus efficace (en temps d'exécution). Qui plus est, la fonction `somme` reçoit déjà implicitement un vecteur par référence, de sorte qu'il n'y a aucune raison de lui transmettre différemment le second vecteur.

Le même raisonnement s'applique à la fonction de calcul du produit scalaire (que nous nommons `prodscal`).

En ce qui concerne la valeur de retour de `somme`, également de type `vecteur3d`, il n'est en revanche pas possible de la transmettre par référence. En effet, ce « résultat » (de type `vecteur3d`) sera créé au sein de la fonction elle-même, ce qui signifie que l'objet correspondant sera de classe automatique, donc détruit à la fin de l'exécution de la fonction. Il faut donc absolument en transmettre la valeur.

Voici ce que pourrait être la déclaration de notre classe `vecteur3d` (ici, seul le constructeur a été prévu en ligne) :

```
    /* déclaration de la classe vecteur3d */
class vecteur3d
{
    float x, y, z ;
  public :
      vecteur3d (float c1=0, float c2=0, float c3=0)   //
constructeur
     {  x=c1 ; y=c2 ; z=c3;
     }
    vecteur3d somme (vecteur3d &) ;       // somme (résultat
par valeur)
    float prodscal (vecteur3d &) ;       // produit scalaire
     void affiche () ;                          // affichage
composantes
} ;
```

Voici sa définition :

```
    /* définition de la classe vect3d */

#include <iostream>
using namespace std ;
vecteur3d vecteur3d::somme (vecteur3d & v)
{   vecteur3d res ;
    res.x = x + v.x ;
    res.y = y + v.y ;
    res.z = z + v.z ;
```

```
    return res ;
}
float vecteur3d::prodscal (vecteur3d & v)
{   return ( v.x * x + v.y * y + v.z * z) ;
}
void vecteur3d::affiche ()
{   cout << "< " << x << ", " << y << ", " << z << ">" ;
}
```

Voici un petit programme d'essai de la classe `vecteur3d`, accompagné des résultats produits par son exécution :

```
   /* programme d'essai de la classe vecteur3d */
#include <iostream>   // voir N.B. du paragraphe Nouvelles possibilités
                      // d'entrées-sorties du chapitre 2
using namespace std ;
main()
{  vecteur3d v1 (1,2,3), v2 (3,0, 2), w ;
   cout << "v1 = " ; v1.affiche () ; cout << "\n" ;
   cout << "v2 = " ; v2.affiche () ; cout << "\n" ;
   cout << "w = " ;  w.affiche () ;  cout << "\n" ;
   w = v1.somme (v2) ;
   cout << "w = " ;  w.affiche () ;  cout << "\n" ;
   cout << "V1.V2 = " << v1.prodscal (v2) << "\n" ;
}
```

```
v1 = < 1, 2, 3>
v2 = < 3, 0, 2>
w = < 0, 0, 0>
w = < 4, 2, 5>
V1.V2 = 9
```

Exercice 75

Énoncé

Comment pourrait-on adapter la classe `point` créée dans l'exercice 68, pour qu'elle dispose d'une fonction membre `compte` fournissant le nombre

de points existant à un instant donné ?

Solution

On pourrait certes introduire une fonction membre classique. Toutefois, cette solution présenterait l'inconvénient d'obliger l'utilisateur à appliquer une telle fonction à un objet de type `point`. Que penser alors d'un appel tel que `p.compte()` (`p` étant un `point`) pour connaître le nombre de points ? Qui plus est, comment faire appel à `compte` s'il n'existe aucun point ?

La solution la plus agréable consiste à faire de `compte` une fonction statique. On la déclarera donc ainsi :

```
static int compte () ;
```

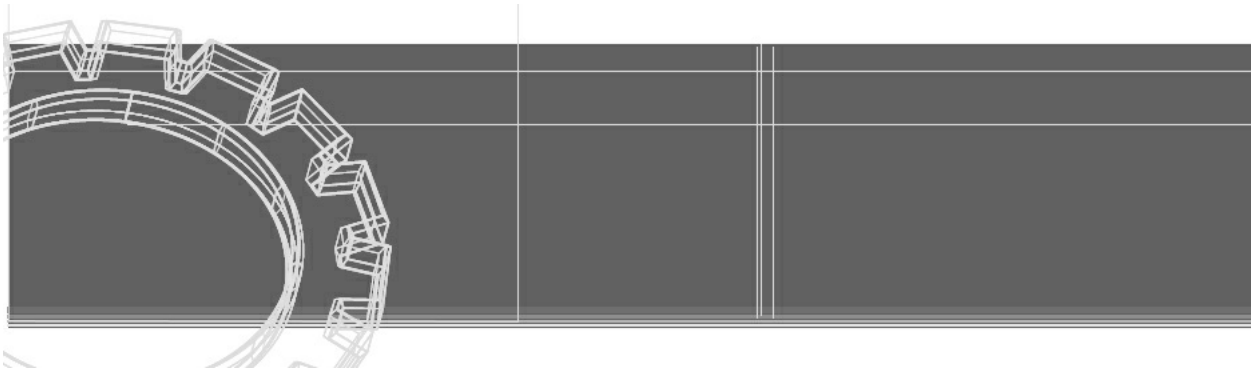
Voici sa définition :

```
int point::compte ()
{  return nb_pts ;
}
```

Voici un exemple d'appel de `compte` au sein d'un programme dans lequel la classe `point` a été déclarée :

```
cout << "il y a " << point::compte () << "points\n" ;
```

Chapitre 9
Construction, destruction et initialisation des objets



Rappels

Appels du constructeur et du destructeur

Dans tous les cas (objets statiques, automatiques ou dynamiques), s'il y a appel du **constructeur**, celui-ci a lieu après l'allocation de l'emplacement mémoire destiné à l'objet. De même, s'il existe un **destructeur**, ce dernier est appelé avant la libération de l'espace mémoire associé à l'objet.

Les objets automatiques et statiques

Les objets **automatiques** sont créés par une déclaration soit dans une fonction, soit au sein d'un bloc. Ils sont créés au moment de l'exécution de la déclaration (laquelle, en C++, peut apparaître n'importe où dans un programme). Ils sont détruits lorsqu'on sort de la fonction ou du bloc.

Les objets **statiques** sont créés par une déclaration située en dehors de toute fonction ou par une déclaration précédée du mot-clé `static` (dans une fonction ou dans un bloc). Ils sont créés avant l'entrée dans la fonction `main` et détruits après la fin de son exécution.

Les objets temporaires

L'appel explicite, au sein d'une expression, du constructeur d'un objet provoque la création d'un objet temporaire (on n'a pas accès à son adresse) qui pourra être automatiquement détruit dès qu'il ne sera plus utile. Par exemple, si une classe `point` possède le constructeur `point (float, float)` et si `a` est de type `point`, nous pouvons écrire :

```
a = point (1.5, 2.25) ;
```

Ne confondez pas une telle affectation avec l'initialisation d'un objet lors de sa déclaration.

Cette instruction provoque la création d'un objet temporaire de type `point` (avec appel du constructeur concerné), suivie de l'affectation de cet objet à `a`.

Les objets dynamiques

Ils sont créés par l'opérateur `new`**,** auquel on doit fournir, le cas échéant, les valeurs des arguments destinés à un constructeur, comme dans cet exemple (qui suppose qu'il existe une classe `point` possédant le constructeur `point (float, float))` :

```
point * adp ;
   .....
adp = new point (2.5, 5.32) ; // création d'un objet de type
point, par
                                  // appel d'un constructeur à
deux arguments
```

Comme pour les variables ordinaires, on peut préciser un nombre d'objets, mais, des restrictions apparaissent alors, qui portent sur le constructeur qu'il est possible d'appeler (voyez ci-après la rubrique « tableaux d'objets »).

L'accès aux membres d'un objet dynamique est réalisé comme pour les variables ordinaires. Par exemple, si `point` possède une méthode nommée `affiche`, on pourra l'appeler par `(*adp).affiche ()` ou encore par `adp->affiche ()`.

Les objets dynamiques n'ont pas de durée de vie définie a priori. Ils sont détruits à la demande en utilisant l'opérateur `delete` comme dans : `delete adr`. (Dans le cas d'un tableau d'objets, la syntaxe sera différente, voir un peu plus loin.)

Construction d'objets contenant des objets membre

Une classe peut posséder un membre donnée qui est lui-même de type classe. En voici un exemple. Si nous avons défini :

```
class point
{   float x, y ;
   public :
    point (float, float) ;
    .....
} ;
```

nous pouvons définir une classe `pointcol`, dont un membre est de type `point` :

```
class pointcol
{   point p ;
    int couleur ;
   public :
    pointcol (float, float, int) ;
    .....
} ;
```

Dans ce cas, lors de la création d'un objet de type `pointcol`, il y aura tout

d'abord appel d'un constructeur de `pointcol`, puis appel d'un constructeur de `point` ; ce dernier recevra les arguments qu'on aura mentionnés dans l'en-tête de la définition du constructeur de `pointcol`. Par exemple, avec :

```
pointcol::pointcol (float abs, float ord, int coul) : p (abs,
ord)
{   ...
}
```

on précise que le constructeur du membre `p` recevra en argument les valeurs `abs` et `ord`.

Si l'en-tête de `pointcol` ne mentionnait rien concernant `p`, il faudrait alors que le type `point` possède un constructeur sans argument pour que cela soit correct.

Initialisation d'objets et constructeur par recopie

En C++, on parle d'initialisation d'un objet dans des situations telles que :

```
point a = 5 ;     // il doit exister un constructeur à un
argument de type entier
point b = a ;     // il doit exister un constructeur à un
argument de type point
```

Le deuxième cas correspond à l'initialisation d'un objet à l'aide d'un autre objet de même type, qu'on nomme parfois « initialisation par recopie ». L'opération est réalisée par appel de ce que l'on nomme un « constructeur par recopie ».

Mais il existe d'autres situations, plus courantes, qui mettent en œuvre un tel mécanisme, à savoir :

- la transmission d'un objet par valeur en argument d'appel d'une fonction ;
- la transmission d'un objet par valeur en valeur de retour d'une fonction.

Dans toutes ces situations d'initialisation par recopie, le constructeur par recopie employé est :

- soit un constructeur de la forme `type (type &)` ou mieux `type (const type &)` s'il en existe un (les deux formes peuvent exister) ; ce dernier doit alors prendre en charge la recopie de tous les membres de l'objet, y

compris ceux qui sont des objets ; il peut cependant s'appuyer sur les possibilités de transmission d'information entre constructeurs, présentée auparavant ;

La transmission par référence est obligatoire ici. D'autre part, le fait d'utiliser l'attribut `const` permet d'appliquer le constructeur à un objet constant ou à une expression ; rappelons que, dans ce cas, il y création d'un objet temporaire dont on transmet la référence au constructeur.

- soit, dans le cas contraire, ce que l'on nomme un « constructeur de recopie par défaut », qui recopie les différents membres de l'objet. Si certains de ces membres sont eux-mêmes des objets, la recopie sera réalisée par appel de son propre constructeur par recopie (qui pourra être soit un constructeur par défaut, soit un constructeur défini dans la classe correspondante).

Les tableaux d'objets

Si `point` est un type objet possédant un constructeur sans argument (ou, situation généralement déconseillée, sans constructeur), la déclaration :

```
point courbe [20] ;
```

crée un tableau `courbe` de 20 objets de type `point` en appelant, le cas échéant, le constructeur pour chacun d'entre eux. Notez toutefois que `courbe` n'est pas lui-même un objet.

De même :

```
point * adcourbe = new point [20] ;
```

alloue l'emplacement mémoire nécessaire à vingt objets (consécutifs) de type `point`, en appelant, le cas échéant, le constructeur pour chacun d'entre eux, puis place l'adresse du premier dans `adcourbe`.

La destruction d'un tableau d'objets se fait en utilisant une syntaxe particulière de l'opérateur `new`. Par exemple, pour détruire le tableau précédent, on écrira :

```
delete [] adcourbe ;
```

Remarque

En théorie, on peut compléter la déclaration d'un tableau d'objets par un initialiseur contenant une liste de valeurs (elles peuvent éventuellement être de types différents). Chaque valeur est alors transmise à un constructeur approprié. Ces valeurs doivent être constantes pour les tableaux de classe statique ; il peut s'agir d'expressions pour les tableaux de classe automatique. On peut ne pas préciser de valeurs pour les derniers éléments du tableau.

On notera qu'un tel initialiseur ne peut pas être utilisé avec des tableaux dynamiques (il en va de même pour les tableaux ordinaires !).

Exercice 76

Énoncé

Comment concevoir le type classe `chose` de façon que ce petit programme :

```
main()
{  chose x ;
   cout << "bonjour\n" ;
}
```

fournisse les résultats suivants :

```
création objet de type chose
bonjour
destruction objet de type chose
```

Que fournira alors l'exécution de ce programme (utilisant le même type `chose`) :

```
main()
{ chose * adc = new chose
}
```

Solution

Il suffit de prévoir, dans le constructeur de `chose`, l'instruction :

```
cout << "création objet de type chose\n" ;
```

et, dans le destructeur, l'instruction :

```
cout << "destruction objet de type chose\n" ;
```

Dans ce cas, le deuxième programme fournira simplement à l'exécution (puisqu'il crée un objet de type `chose` sans jamais le détruire) :

```
création objet de type chose
```

Exercice 77

Énoncé

Quels seront les résultats fournis par l'exécution du programme suivant (ici, la déclaration de la classe `demo`, sa définition et le programme d'utilisation ont été regroupés en un seul fichier) :

```
#include <iostream>
using namespace std ;
class demo
{  int x, y ;
  public :
   demo (int abs=1, int ord=0)   // constructeur I (0, 1 ou
2 arguments)
     { x = abs ;  y = ord ;
       cout << "constructeur I            : " << x << " " <<
y << "\n" ;
     }
    demo (demo &) ;                 // constructeur II (par
recopie)
   ~demo () ;                    // destructeur
} ;

demo::demo (demo & d)              // ou demo::demo (const
demo & d)
```

```
{  cout << "constructeur II (recopie) : " << d.x << " " <<
d.y << "\n" ;
   x = d.x ;  y = d.y ;
}
demo::~demo ()
{  cout << "destruction               : " << x << " " << y
<< "\n" ;
}

main ()
{
     void  fct  (demo,  demo  *)  ;        //  proto  fonction
indépendante fct
   cout << "début main\n" ;
   demo a ;
   demo b = 2 ;
   demo c = a ;
   demo * adr = new demo (3,3) ;
   fct (a, adr) ;
   demo d = demo (4,4) ;
   c = demo (5,5) ;
   cout << "fin main\n" ;
}
void fct (demo d, demo * add)
{   cout << "entrée fct\n" ;
    delete add ;
    cout << "sortie fct\n" ;
}
```

Voici les résultats que fournit le programme (ici, nous avons utilisé le compilateur C++ Builder X ; ce point n'ayant d'importance que pour les objets temporaires, dans le cas des implémentations qui ne respecteraient pas la norme quant à l'instant de leur destruction).

La plupart des lignes sont assorties d'explications complémentaires présentées conventionnellement sous forme de commentaires et qui précisent les instructions concernées.

```
début main
constructeur I            : 1 0   /* demo a ;          */
constructeur I            : 2 0   /* demo b = 2 ;      */
constructeur II (recopie) : 1 0   /* demo c = a ;      */
constructeur I            : 3 3   /* new demo (3, 3)   */
```

```
constructeur II (recopie) : 1 0   /* recopie de la valeur de a
dans fct(a, ...)    */
                                  /*   ce qui crée un objet
temporaire             */
entrée fct
destruction                 : 3 3   /* delete add ;   (dans
fct)                   */
sortie fct
destruction                    : 1 0    /* destruction objet
temporaire créé pour        */
                                  /*  l'appel de fct
                       */
constructeur I             : 4 4   /* demo d = demo(4, 4)
                    */
constructeur  I                  : 5 5    /*  c  =  demo(5,  5)
(contruction objet temporaire) */
destruction                    : 5 5    /* destruction objet
temporaire précédent        */
fin main
destruction               : 4 4   /* destruction d */
destruction               : 5 5   /* destruction c */
destruction               : 2 0   /* destruction b */
destruction               : 1 0   /* destruction a */
```

Notez bien que l'affectation `c = demo (5,5)` entraîne la création d'un objet temporaire par appel du constructeur de `demo` (arguments `5` et `5`) ; cet objet est ensuite affecté à `a`. On constate d'ailleurs que cet objet est effectivement détruit aussitôt après. Mais il existe certaines implémentations qui ne respectent pas la norme et où cela peut se produire plus tard.

Par ailleurs, l'appel de `fct` a entraîné la construction d'un objet temporaire, par appel du constructeur par recopie. Cet objet est ici libéré dès la sortie de la fonction. Là encore, dans certaines implémentations, cela peut se produire plus tard.

Exercice 78

Énoncé

Créer une classe `point` ne contenant qu'un constructeur sans arguments, un destructeur et un membre donnée privé représentant un numéro de point (le

premier créé portera le numéro 1, le suivant le numéro 2...). Le constructeur affichera le numéro du point créé et le destructeur affichera le numéro du point détruit. Écrire un petit programme d'utilisation créant dynamiquement un tableau de 4 points et le détruisant.

Solution

Pour pouvoir numéroter nos points, il nous faut pouvoir compter le nombre de fois où le constructeur a été appelé, ce qui nous permettra bien d'attribuer un numéro différent à chaque point. Pour ce faire, nous définissons, au sein de la classe `point`, un membre donnée statique `nb_points`. Ici, il sera incrémenté par le constructeur mais le destructeur n'aura pas d'action sur lui. Comme tout membre statique, `nb_points` devra être initialisé.

Voici la déclaration (définition) de notre classe, accompagnée du programme d'utilisation demandé :

```
#include <iostream>
using namespace std ;
class point
{   int num;
    static int nb_points ;
  public :
    point ()
       {  num = ++nb_points ;
             cout << "création point numéro     : " << num <<
"\n" ;
       }
    ~point ()
         {   cout << "Destruction point numéro : " << num <<
"\n" ;
       }
} ;
int point::nb_points=0 ;     // initialisation obligatoire
main()
{  point * adcourb = new point [4] ;
   delete [] adcourb ;
}
```

À titre indicatif, voici les résultats fournis par ce programme :

```
création point numéro    : 1
création point numéro    : 2
création point numéro    : 3
```

```
création point numéro    : 4
Destruction point numéro : 4
Destruction point numéro : 3
Destruction point numéro : 2
Destruction point numéro : 1
```

Exercice 79

Énoncé

1. Réaliser une classe nommée `set_int` permettant de manipuler des ensembles de nombres entiers. On devra pouvoir réaliser sur un tel ensemble les opérations classiques suivantes : lui ajouter un nouvel élément, connaître son cardinal (nombre d'éléments), savoir si un entier donné lui appartient.

Ici, on conservera les différents éléments de l'ensemble dans un tableau alloué dynamiquement par le constructeur. Un argument (auquel on pourra prévoir une valeur par défaut) lui précisera le nombre maximal d'éléments de l'ensemble.

2. Écrire, en outre, un programme (`main`) utilisant la classe `set_int` pour déterminer le nombre d'entiers différents contenus dans 20 entiers lus en données.

3. Que faudrait-il faire pour qu'un objet du type `set_int` puisse être transmis par valeur, soit comme argument d'appel, soit comme valeur de retour d'une fonction ?

N.B. Le chapitre 17 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solutions

1. La déclaration de la classe découle de l'énoncé :

```
/*          fichier SETINT1.H        */
```

```
        /* déclaration de la classe set_int */
class set_int
{
    int * adval ;                   // adresse du tableau des
valeurs
   int nmax ;                  // nombre maxi d'éléments
   int nelem ;                 // nombre courant d'éléments

public :
   set_int (int = 20) ;        // constructeur
   ~set_int () ;               // destructeur
   void ajoute (int) ;         // ajout d'un élément
   int appartient (int) ;      // appartenance d'un élément
   int cardinal () ;           // cardinal de l'ensemble
} ;
```

Le membre donnée `adval` est destiné à pointer sur le tableau d'entiers qui sera alloué par le constructeur. Le membre `nmax` représentera la taille de ce tableau, tandis que `nelem` fournira le nombre effectif d'entiers stockés dans ce tableau. Ces entiers seront, cette fois, rangés dans l'ordre où ils seront fournis à `ajoute`, et non plus à un emplacement prédéterminé, comme nous l'avions fait pour les caractères (dans les exercices du chapitre précédent).

Comme la création d'un objet entraîne ici l'allocation dynamique d'un emplacement mémoire, il est raisonnable de prévoir la libération de cet emplacement lors de la destruction de l'objet ; cette opération doit donc être prise en charge par le destructeur, d'où la présence de cette fonction membre.

Voici la définition de notre classe :

```
#include "setint1.h"
set_int::set_int (int dim)
{  adval = new int [nmax = dim] ;    // allocation tableau de
valeurs
   nelem = 0 ;
}
set_int::~set_int ()
{  delete [] adval ;                 // libération tableau de
valeurs
}
void set_int::ajoute (int nb)
{   // on examine si nb appartient déjà à l'ensemble
    // en utilisant la fonction membre appartient
      // s'il n'y appartient pas et si l'ensemble n'est pas
plein
    // on l'ajoute
    if (!appartient (nb) && (nelem<nmax))  adval [nelem++] =
```

```
nb ;
}
int set_int::appartient (int nb)
{  int i=0 ;
     // on examine si nb appartient déjà à l'ensemble
       // (si ce n'est pas le cas, i vaudra nele en fin de
boucle)
   while ( (i<nelem) && (adval[i] != nb) ) i++ ;
   return (i<nelem) ;
}
int set_int::cardinal ()
{  return nelem ;
}
```

Notez que, dans la fonction membre `ajoute`, nous avons utilisé la fonction membre `appartient` pour vérifier que le nombre à ajouter ne figurait pas déjà dans notre ensemble.

Par ailleurs, l'énoncé ne prévoit rien pour le cas où l'on cherche à ajouter un élément à un ensemble déjà « plein » ; ici, nous nous sommes contenté de ne rien faire dans ce cas. Dans la pratique, il faudrait soit prévoir un moyen pour que l'utilisateur soit prévenu de cette situation, soit, mieux, prévoir automatiquement l'agrandissement de la zone dynamique associée à l'ensemble.

2. Voici le programme d'utilisation demandé :

```
#include "setint1.h"
#include <iostream>
using namespace std ;
main()
{  set_int ens ;
   cout << "donnez 20 entiers \n" ;
   int i, n ;
   for (i=0 ; i<20 ; i++)
      { cin >> n ;
        ens.ajoute (n) ;
      }
    cout << "il y a : " << ens.cardinal () << " entiers
différents\n" ;
}
```

À titre indicatif, voici un exemple d'exécution :

```
donnez 20 entiers
0 2 5 2 8 5 1 8 2 0 7 2 5 5 4 5 0 0 4 5
il y a : 7 entiers différents
```

3. Telle qu'est actuellement prévue notre classe `set_int`, si un objet de ce type est transmis par valeur, soit en argument d'appel, soit en retour d'une fonction, il y a appel du constructeur de recopie par défaut. Or, ce dernier se contente d'effectuer une copie des membres donnée de l'objet concerné, ce qui signifie qu'on se retrouve en présence de deux objets contenant deux pointeurs différents sur un même tableau d'entiers. Un problème va donc se poser, dès lors que l'objet copié sera détruit (ce qui peut se produire dès la sortie de la fonction) ; en effet, dans ce cas, le tableau dynamique d'entiers sera détruit, alors même que l'objet d'origine continuera à « pointer » dessus.

Indépendamment de cela, d'autres problèmes similaires pourraient se poser si la fonction était amenée à modifier le contenu de l'ensemble ; en effet, on modifierait alors le tableau d'entiers original, chose à laquelle on ne s'attend pas dans le cas de la transmission par valeur.

Pour régler ces problèmes, il est nécessaire de munir notre classe d'un constructeur par recopie approprié, c'est-à-dire tenant compte de la « partie dynamique » de l'objet (on parle parfois de « copie profonde »). Pour ce faire, on alloue un second emplacement pour un tableau d'entiers, dans lequel on recopie les valeurs du premier ensemble. Naturellement, il ne faut pas oublier de procéder également à la recopie des membres donnée, puisque celle-ci n'est plus assurée par le constructeur de recopie par défaut (lequel n'est plus appelé, dès lors qu'un constructeur par recopie a été défini).

Nous ajouterons donc dans la déclaration de notre classe :

```
set_int (set_int &) ;                    // constructeur par
recopie
```

Et, dans sa définition :

```
set_int::set_int (set_int & e)     // ou set_int::set_int
                                   // (const set_int & e)
{
    adval = new int [nmax = e.nmax] ; // allocation nouveau
tableau
   nelem = e.nelem ;
   int i ;
   for (i=0 ; i<nelem ; i++)          // copie ancien tableau
dans nouveau
   adval[i] = e.adval[i] ;
```

```
}
```

Exercice 80

Énoncé

Modifier l'implémentation de la classe précédente (avec son constructeur par recopie) de façon que l'ensemble d'entiers soit maintenant représenté par une **liste chaînée** (chaque entier est rangé dans une structure comportant un champ destiné à contenir un nombre et un champ destiné à contenir un pointeur sur la structure suivante). L'interface de la classe (la partie publique de sa déclaration) devra rester inchangée, ce qui signifie qu'un client de la classe continuera à l'employer de la même façon.

Solution

Comme nous le suggère l'énoncé, nous allons donc définir une structure que nous nommerons élément :

```
struct noeud
{   int valeur ;                    // valeur d'un élément de
l'ensemble
   noeud * suivant ;         // pointeur sur le nœud suivant de
la liste
} ;
```

Notre structure `noeud` peut être définie indifféremment dans la déclaration de la classe `set_int` ou en dehors.

En ce qui concerne les membres donnée privés de la classe, nous ne conserverons que `nelem` qui, bien que non indispensable, nous évitera de parcourir toute la liste pour déterminer le cardinal de notre ensemble.

En revanche, nous y introduirons un pointeur nommé `debut`, destiné à contenir l'adresse du premier élément de la liste, s'il existe (au départ, il sera initialisé à `NULL` ou a `nullptr` depuis C++11).

En ce qui concerne le constructeur de `set_int`, nous lui conserverons son

argument (de type `int`), bien qu'ici il n'ait plus aucun intérêt, et cela dans le but de ne pas modifier l'interface de notre classe (comme le demandait l'énoncé).

Voici donc la nouvelle déclaration de notre classe :

```
          /*         fichier SETINT3.H        */
          /* déclaration de la classe set_int */
  struct noeud
  {
      int valeur ;                  // valeur d'un élément de
  l'ensemble
     noeud * suivant ;          // pointeur sur le nœud suivant
  de la liste
  } ;

  class set_int
  {
     noeud * debut ;             // pointeur sur le début de la
  liste
     int nelem ;               // nombre courant d'éléments
   public :
      set_int (int = 20) ;         // constructeur   (argument
  inutile ici)
     set_int (set_int &) ;     // constructeur par recopie
     ~set_int () ;             // destructeur
     void ajoute (int) ;       // ajout d'un élément
     int appartient (int) ;    // appartenance d'un élément
     int cardinal () ;         // cardinal de l'ensemble
  } ;
```

La définition du nouveau constructeur ne présente pas de difficulté. La fonction membre `ajoute` réalise une classique insertion d'un `noeud` en début de liste et incrémente le nombre d'éléments de l'ensemble. La fonction `appartient` effectue une exploration de liste tant qu'elle n'a pas trouvé la valeur concernée ou atteint la fin de la liste. En revanche, le nouveau constructeur par recopie doit recopier la liste chaînée. Il réalise à la fois une exploration de la liste d'origine et une insertion dans la liste copiée. Quant au destructeur, il doit maintenant libérer systématiquement tous les emplacements des différents nœuds créés pour la liste.

Voici la nouvelle définition de notre classe :

```
  #include <stdlib.h>          // pour NULL
  #include "setint3.h"
  set_int::set_int (int dim)     // dim est conservé pour la
```

```
compatibilité
                                // avec l'ancienne classe
{  debut = NULL ;               // ou nullptr depuis C++11
   nelem = 0 ;
}
set_int::set_int (set_int & e)     // ou :   set_int::set_int
(const set_int & e)
{  nelem = e.nelem ;
       // création d'une nouvelle liste identique à l'ancienne
   noeud * adsource = e.debut  ;
   noeud * adbut ;
   debut = NULL ;
   while (adsource)
         { adbut = new noeud ;                    // création
nouveau nœud
         adbut->valeur = adsource->valeur ; // copie valeur
           adbut->suivant = debut ;              // insertion
nouveau nœud
          debut = adbut ;                     // dans nouvelle
liste
           adsource = adsource->suivant ;      // nœud suivant
ancienne liste
       }
}
set_int::~set_int ()
{   noeud * adn ;
    noeud * courant = debut ;
    while (courant)
  { adn = courant ;                  // libération de tous
    courant = courant->suivant ;     // les nœuds
    delete [] adn ;                  // de la liste
  }
}
void set_int::ajoute (int nb)
{ if (!appartient (nb) )           // si nb n'appartient pas à
la liste
     { noeud * adn = new noeud ;   // on l'ajoute en début de
liste
       adn->valeur = nb ;
       adn->suivant = debut ;
       debut = adn ;
       nelem++ ;
     }
}
int set_int::appartient (int nb)
{  noeud * courant = debut ;
       // attention à l'ordre des deux conditions
     while (courant && (courant->valeur != nb) ) courant =
```

```
courant->suivant ;
   return (courant != NULL) ;
}

int set_int::cardinal ()
{  return nelem ;
}
```

Notez que le programme d'utilisation proposé dans l'exercice 26 reste valable ici, puisque nous n'avons précisément pas modifié l'interface de notre classe.

Par ailleurs, le problème évoqué à propos de l'ajout d'un élément à un ensemble « plein » ne se pose plus ici, compte tenu de la nouvelle implémentation de notre classe (hormis un éventuel manque de mémoire).

Exercice 81

Énoncé

Modifier la classe `set_int` précédente (implémentée sous la forme d'une liste chaînée, avec ou sans son constructeur par recopie) pour qu'elle dispose de ce que l'on nomme un « itérateur » sur les différents éléments de l'ensemble. Rappelons qu'il s'agit d'un mécanisme permettant d'accéder séquentiellement aux différents éléments de l'ensemble. On prévoira trois nouvelles fonctions membre : `init`, pour initialiser le processus d'itération ; `prochain`, pour fournir l'élément suivant lorsqu'il existe et `existe`, pour tester s'il existe encore un élément non exploré.

On complétera alors le programme d'utilisation précédent (en fait, celui de l'exercice 26), de manière qu'il affiche les différents entiers contenus dans les valeurs fournies en donnée.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici. D'autre part, cet exercice sera plus profitable s'il est traité après l'exercice du chapitre 3 qui proposait l'introduction d'un tel itérateur dans une classe représentant des ensembles de caractères (mais dont l'implémentation était différente de l'actuelle classe).

Solution

Ici, la gestion du mécanisme d'itération nécessite l'emploi d'un pointeur (que nous nommerons `courant`) sur un nœud de notre liste. Nous conviendrons qu'il pointe sur le premier élément non encore traité dans l'itération, c'est-à-dire dont la valeur correspondante n'a pas encore été renvoyée par la fonction `prochain`. Il n'est pas utile, ici, de prévoir un membre donnée pour indiquer si la fin de liste a été atteinte ; en effet, avec la convention adoptée, il nous suffit de tester la valeur de `courant` (qui sera égale à NULL, lorsque l'on sera en fin de liste).

Le rôle de la fonction `init` se limite à l'initialisation de `courant` à la valeur du pointeur sur le début de la liste (`debut`).

La fonction `suivant` fournira en retour la valeur entière associée au nœud pointé par `courant` lorsqu'il existe (`courant` différent de `NULL`) ou la valeur `0` dans le cas contraire (il s'agit, là encore, d'une convention destinée à protéger l'utilisateur ayant appelé cette fonction alors que la fin de liste était déjà atteinte et, donc, qu'aucun élément de l'ensemble n'était disponible).

De plus, dans le premier cas (usuel), la fonction `suivant` actualisera la valeur de `courant`, de manière qu'il pointe sur le nœud suivant.

Enfin, la fonction `existe` examinera simplement la valeur de `debut` pour savoir s'il existe encore un élément à traiter.

Voici la déclaration complète de notre nouvelle classe :

```
          /*         fichier SETINT4.H        */
          /* déclaration de la classe set_int */
  struct noeud
  {   int valeur ;                  // valeur d'un élément de
  l'ensemble
     noeud * suivant ;          // pointeur sur le nœud suivant
  de la liste
  } ;
  class set_int
  {  noeud * debut ;             // pointeur sur le début de la
  liste
     int nelem ;               // nombre courant d'éléments
     noeud * courant ;         // pointeur sur nœud courant
   public :
     set_int (int = 20) ;      // constructeur
     set_int (set_int &) ;     // constructeur par recopie
```

```
   ~set_int () ;             // destructeur
   void ajoute (int) ;       // ajout d'un élément
   int appartient (int) ;    // appartenance d'un élément
   int cardinal () ;         // cardinal de l'ensemble
   void init () ;            // initialisation itération
   int prochain () ;         // entier suivant
   int existe () ;           // test fin liste
} ;
```

Voici la définition des trois nouvelles fonctions membre `init`, `suivant` et `existe` :

```
void set_int::init ()
{  courant = debut ;
}
int set_int::prochain ()
{  if (courant)
      { int val = courant->valeur ;
        courant = courant->suivant ;
        return val ;
      }
     else return 0 ;    // par convention
}
int set_int::existe ()
{  return (courant != NULL) ;
}
```

Voici le nouveau programme d'utilisation demandé :

```
   /* utilisation de la classe set_int */
#include "setint1.h"
#include <iostream>
using namespace std ;
main()
{  set_int ens ;
   cout << "donnez 20 entiers \n" ;
   int i, n ;
   for (i=0 ; i<20 ; i++)
      { cin >> n ;
        ens.ajoute (n) ;
      }
     cout << "il y a : " << ens.cardinal () << " entiers
différents\n" ;
   cout << "Ce sont : \n" ;
   ens.init () ;
   while (ens.existe ())
      cout << ens.prochain() << " " ;
}
```

À titre indicatif, voici un exemple d'exécution :

```
donnez 20 entiers
0 2 1 5 4 1 2 0 2 1 4 5 1 2 0 2 1 4 5 2
il y a : 5 entiers différents
Ce sont :
4 5 1 2 0
```

Exercice 82

Énoncé

Quels seront les résultats fournis par l'exécution de ce programme :

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
 public :
  point ()
    { x=0 ; y=0 ;
      cout << "** constructeur 0 argument\n" ;
    }
  point (int abs)
    { x=abs ; y=0 ;
      cout << "** constructeur 1 argument\n" ;
    }
  point (int abs, int ord)
    { x=abs ; y=ord ;
      cout << "** constructeur 2 arguments\n" ;
    }
  point (const point & p)
    { x=p.x ; y=p.y ;
      cout << "**constructeur par recopie\n" ;
    }
  void affiche ()
    { cout << "point de coordonnées : " << x << " " << y <<
"\n" ;
    }
} ;
main()
{ point a(10,20) ;
  point b(30,40) ;
```

```
    point courbe[6] = { 4, a, 0, b} ;
    for (int i=0 ; i<6 ; i++) courbe[i].affiche() ;
  }
```

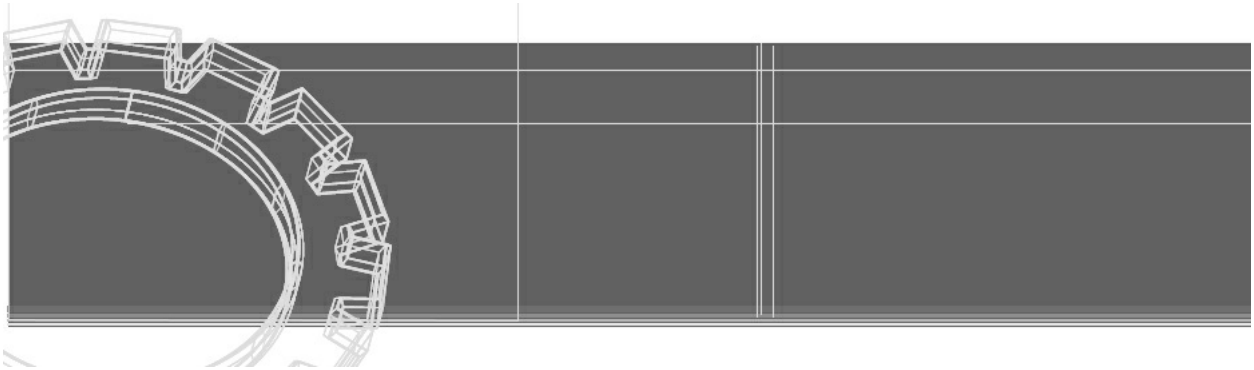
```
** constructeur 2 arguments     // construction (classique) de
a
** constructeur 2 arguments     // construction (classique) de
b
** constructeur 1 argument     // construction courbe[0]
**constructeur par recopie     // construction courbe[1]
** constructeur 1 argument     // construction courbe[2]
**constructeur par recopie     // construction courbe[3]
** constructeur 0 argument     // construction courbe[4]
** constructeur 0 argument     // construction courbe[5]
point de coordonnées : 4 0
point de coordonnées : 10 20
point de coordonnées : 0 0
point de coordonnées : 30 40
point de coordonnées : 0 0
point de coordonnées : 0 0
```

Après la construction classique des points `a` et `b`, il y a création d'un tableau de 6 objets de type `point`. Les quatre premiers points sont construits par :

- appel d'un constructeur à un argument de type `int` (valeur `4`) pour le premier point ;
- appel d'un constructeur à un argument de type `point` (valeur `a`) pour le deuxième ;
- appel d'un constructeur à un argument de type `int` (valeur `0`) pour le troisième ;
- appel d'un constructeur à un argument de type `point` (valeur `b`) pour le quatrième.

Les deux derniers points du tableau ne disposent pas d'initialiseur ; ils sont donc construits par appel d'un constructeur sans argument.

Chapitre 10
Les fonctions amies



Rappels

En C++, l'unité de protection est la classe, et non pas l'objet. Cela signifie qu'une fonction membre d'une classe peut accéder à tous les membres privés de n'importe quel objet de sa classe. En revanche, ces membres privés restent inaccessibles à n'importe quelle fonction membre d'une autre classe ou à n'importe quelle fonction indépendante.

La notion de fonction amie, ou plus exactement de « déclaration d'amitié », permet de déclarer dans une classe les fonctions que l'on autorise à accéder à ses membres privés (données ou fonctions). Il existe plusieurs situations d'amitié.

Fonction indépendante, amie d'une classe A

```
class A
{
   ......
   friend --- fct (-----) ;
```

```
   .....
}
```

La fonction `fct` ayant le prototype spécifié est autorisée à accéder aux membres privés de la classe `A`.

Fonction membre d'une classe B, amie d'une autre classe A

```
class A
{
   .....
   friend --- B:fct (-----) ;
   .....
} ;
```

La fonction `fct`, membre de la classe `B`, ayant le prototype spécifié, est autorisée à accéder aux membres privés de la classe `A`.

Pour qu'il puisse compiler convenablement la déclaration de `A`, donc en particulier la déclaration d'amitié relative à `fct`, le compilateur devra connaître la déclaration de `B` (mais pas nécessairement sa définition).

Généralement, la fonction membre `fct` possédera un argument ou une valeur de retour de type `A` (ce qui justifiera sa déclaration d'amitié). Pour compiler sa déclaration (au sein de la déclaration de `A`), il suffira au compilateur de savoir que `A` est une classe ; si sa déclaration n'est pas connue à ce niveau, on pourra se contenter de :

```
class A ;
```

En revanche, pour compiler la définition de `fct`, le compilateur devra posséder les caractéristiques de `A`, donc disposer de sa déclaration.

Toutes les fonctions d'une classe B sont amies d'une autre classe A

Dans ce cas, plutôt que d'utiliser autant de déclarations d'amitié que de fonctions membre, on utilise (dans la déclaration de la classe `A`) la déclaration (globale) suivante :

```
friend class B ;
```

Pour compiler la déclaration de A, on précisera simplement que B est une classe par :

```
class B ;
```

Quant à la déclaration de la classe B, elle nécessitera généralement (dès qu'une de ses fonctions membre possédera un argument ou une valeur de retour de type A) la déclaration de la classe A.

Exercice 83

Énoncé

Soit la classe point suivante :

```
class point
{   int x, y ;
   public :
    point (int abs=0, int ord=0)
      { x = abs ; y = ord ;
      }
} ;
```

Écrire une fonction indépendante affiche, amie de la classe point, permettant d'afficher les coordonnées d'un point. On fournira séparément un fichier source contenant la nouvelle déclaration (définition) de point et un fichier source contenant la définition de la fonction affiche. Écrire un petit programme (main) qui crée un point de classe automatique et un point de classe dynamique et qui en affiche les coordonnées.

Solution

Nous devons donc réaliser une fonction indépendante, nommée affiche, amie de la classe point. Une telle fonction, contrairement à une fonction membre, ne reçoit plus d'argument implicite ; affiche devra donc recevoir un argument de type point. Son prototype sera donc de la forme :

```
void affiche (point) ;
```

si l'on souhaite transmettre un point par valeur, ou de la forme :

```
void affiche (point &) ;
```

si l'on souhaite transmettre un point par référence. Ici, nous choisirons cette dernière possibilité et, comme `affiche` n'a aucune raison de modifier les valeurs du `point` reçu, nous le protégerons à l'aide du qualificatif `const` :

```
void affiche (const point &) ;
```

Le qualificatif `const` permet d'appliquer la fonction `affiche` à un objet constant. Mais elle pourra également être appliquée à une expression de type `point`, voire à une expression d'un type susceptible d'être converti implicitement en un point (voir le chapitre relatif aux conversions définies par l'utilisateur). Ce dernier aspect ne constitue plus nécessairement un avantage !

Manifestement, `affiche` devra pouvoir accéder aux membres privés `x` et `y` de la classe `point`. Il faut donc prévoir une déclaration d'amitié au sein de cette classe, dont voici la nouvelle déclaration :

```
          /*         fichier POINT1.H        */
          /* déclaration de la classe point */
class point
{   int x, y ;
  public :
     friend void affiche (const point &) ;      // const non
obligatoire
    point (int abs=0, int ord=0)
       { x=abs ; y=ord ;
       }
} ;
```

Pour écrire `affiche`, il nous suffit d'accéder aux membres (privés) `x` et `y` de son argument de type `point`. Si ce dernier se nomme `p`, les membres en question se notent (classiquement) `p.x` et `p.y`. Voici la définition de `affiche` :

```
#include "point1.h"              // nécessaire pour compiler
```

```
affiche
#include <iostream>
using namespace std ;
void affiche (const point & p)
{   cout << "Coordonnées : " << p.x << " " << p.y << "\n" ;
}
```

Notez bien que la compilation de `affiche` nécessite la déclaration de la classe `point`, et pas seulement une déclaration telle que `class point`, car le compilateur doit connaître les caractéristiques de la classe `point` (notamment, ici, la localisation des membres `x` et `y`).

Voici le petit programme d'essai demandé :

```
#include "point1.h"
main()
{  point a(1,5) ;
   affiche (a) ;
   point * adp ;
   adp = new point (2, 12) ;
   affiche (*adp) ;              // attention *adp et non adp
}
```

Notez que nous n'avons pas eu à fournir le prototype de la fonction indépendante `affiche`, car il figure dans la déclaration de la classe `point`. Le faire constituerait d'ailleurs une erreur.

Exercice 84

Énoncé

Soit la classe `vecteur3d` définie dans l'exercice 72 par :

```
class vecteur3d
{   float x, y, z ;
   public :
    vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
      { x = c1 ; y = c2 ; z = c3 ; }
     .....
} ;
```

Écrire une fonction indépendante `coincide`, amie de la classe `vecteur3d`,

permettant de savoir si deux vecteurs ont les mêmes composantes (cette fonction remplacera la fonction membre `coincide` qu'on demandait d'écrire dans l'exercice 70).

Si `v1` et `v2` désignent deux vecteurs de type `vecteur3d`, comment s'écrit maintenant le test de coïncidence de ces deux vecteurs ?

Solution

La fonction `coincide` va donc disposer de deux arguments de type `vecteur3d`. Si l'on prévoit de les transmettre par référence, en interdisant leur éventuelle modification dans la fonction, le prototype de `coincide` sera :

```
int coincide (const vecteur3d &, const vecteur3d &) ;
```

Remarque

Là encore, le qualificatif `const` a un double rôle : il permet d'appliquer la fonction `coincide` à des objets constants ou à des expressions de type `point`. Mais il permettra également de l'appliquer à toute valeur susceptible d'être convertie dans le type `point` (voir le chapitre relatif aux conversions définies par l'utilisateur).

Voici la nouvelle déclaration de notre classe :

```
         /*         fichier vect3D.H        */
         /* déclaration de la classe vecteur3d */
class vecteur3d
{   float x, y, z ;
  public :
     friend int coincide (const vecteur3d &, const vecteur3d
&) ;
    vecteur3d (float c1=0, float c2=0, float c3=0)
       {  x = c1 ; y = c2 ; z = c3 ;
       }
} ;
```

Voici la définition de la fonction `coincide` :

```
#include "vect3d.h"              // nécessaire pour compiler
coincide
```

```
int coincide (const vecteur3d & v1, const vecteur3d & v2)
{   if ( (v1.x == v2.x) && (v1.y == v2.y) && (v1.z == v2.z) )
            return 1 ;
     else return 0 ;
}
```

Le test de coïncidence de deux vecteurs s'écrit maintenant :

```
coincide (v1, v2)
```

On notera que, tant dans la définition de `coincide` que dans ce test, on voit se manifester la symétrie du problème, ce qui n'était pas le cas lorsque nous avions fait de `coincide` une fonction membre de la classe `vecteur3d`.

Exercice 85

Énoncé

Créer deux classes (dont les membres donnée sont privés) :

- l'une, nommée `vect`, permettant de représenter des vecteurs à 3 composantes de type `double` ; elle comportera un constructeur et une fonction membre d'affichage ;
- l'autre, nommée `matrice`, permettant de représenter des matrices carrées de dimensions 3 x 3 ; elle comportera un constructeur avec un argument (adresse d'un tableau de 3 x 3 valeurs) qui initialisera la matrice avec les valeurs correspondantes.

Réaliser une fonction indépendante `prod` permettant de fournir le vecteur correspondant au produit d'une matrice par un vecteur. Écrire un petit programme de test. On fournira séparément les deux déclarations de chacune des classes, leurs deux définitions, la définition de `prod` et le programme de test.

Solution

Ici, pour nous faciliter l'écriture, nous représenterons les composantes d'un vecteur par un tableau à trois éléments et les valeurs d'une matrice par un

tableau à 2 indices. La fonction de calcul du produit d'un vecteur par une matrice doit obligatoirement pouvoir accéder aux données des deux classes, ce qui signifie qu'elle devra être déclarée « amie » dans chacune de ces deux classes.

En ce qui concerne ses deux arguments (de type `vect` et `mat`), nous avons choisi la transmission la plus efficace, c'est-à-dire la transmission par référence. Quant au résultat (de type `vect`), il doit obligatoirement être renvoyé par valeur (nous en reparlerons dans la définition de `prod`).

Voici la déclaration de la classe `vect` :

```
                    /* fichier vect1.h */
class matrice ;         // pour pouvoir compiler la déclaration
de vect
class vect
{
    double v[3] ;      // vecteur à 3 composantes
  public :
     vect (double v1=0, double v2=0, double v3=0)        //
constructeur
      { v[0] = v1 ; v[1]=v2 ; v[2]=v3 ; }
      friend vect prod (const matrice &, const vect &) ;//
fonction amie
                                                          //
indépendante
    void affiche () ;
} ;
```

et la déclaration de la classe `mat` :

```
                    /* fichier mat1.h */
class vect ;           // pour pouvoir compiler la déclaration
de matrice
class matrice
{
    double mat[3] [3] ;       // matrice 3 × 3
  public :
      matrice () ;                      // constructeur avec
initialisation à zéro
     matrice (double t [3] [3] ) ; // constructeur à partir
d'un tableau
                                    // 3 × 3
     friend vect prod (const matrice &, const vect &) ;  //
fonction amie
                                                          //
indépendante
```

```
} ;
```

Remarque

Nous avons déclaré constants les arguments de la fonction `vect`, ce qui nous protège d'une éventuelle faute de programmation dans les instructions de cette fonction. Dans ce cas, la fonction pourra être appelée avec en arguments effectifs non seulement des objets constants, mais aussi des expressions d'un type susceptible d'être converti dans le type voulu (comme on le verra dans le chapitre relatif aux conversions définies par l'utilisateur).

La définition des fonctions membre (en fait `affiche`) de la classe `vect` ne présente pas de difficultés :

```
#include "vect1.h"
#include <iostream>
using namespace std ;
void vect::affiche ()
{ int i ;
  for (i=0 ; i<3 ; i++) cout << v[i] << " " ;
  cout << "\n" ;
}
```

Il en va de même pour les fonctions membre (en fait le constructeur) de la classe `mat` :

```
#include "mat1.h"
#include <iostream>
using namespace std ;
matrice::matrice (double t [3] [3])
{
  int i ; int j ;
  for (i=0 ; i<3 ; i++)
    for (j=0 ; j<3 ; j++)
       mat[i] [j] = t[i] [j] ;
}
```

La définition `prod` nécessite les fichiers contenant les déclarations de `vect` et de `mat` :

```
#include "vect1.h"
#include "mat1.h"
```

```
vect prod (const matrice & m, const vect & x)
{
   int i, j ;
   double som ;
   vect res ;       // pour le résultat du produit
   for (i=0 ; i<3 ; i++)
     { for (j=0, som=0 ; j<3 ; j++)
          som += m.mat[i] [j] * x.v[j] ;
       res.v[i] = som ;
     }
   return res ;
}
```

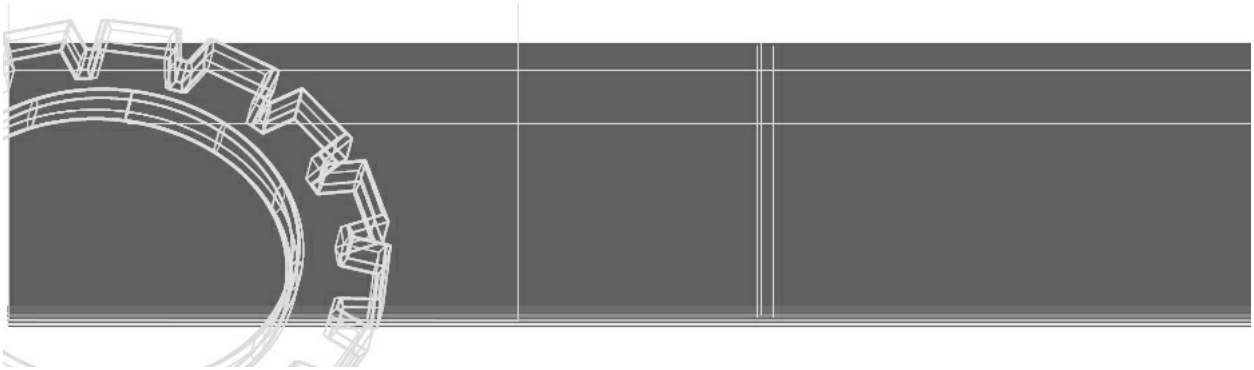
Notez que cette fonction crée un objet automatique `res` de classe `vect`. Il ne peut donc être renvoyé que par valeur. Dans le cas contraire, la fonction appelante récupérerait l'adresse d'un emplacement libéré au moment de la sortie de la fonction.

Voici, enfin, un exemple de programme de test, accompagné de son résultat :

```
#include "vect1.h"
#include "mat1.h"
main()
{  vect w (1,2,3) ;
   vect res  ;
   double tb [3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
   matrice a =  tb  ;
   res = prod(a, w) ;
   res.affiche () ;
}
```

```
 14 32 50
```

Chapitre 11
Surdéfinition d'opérateurs



Rappels

C++ vous permet de surdéfinir les opérateurs existants, c'est-à-dire de leur donner une nouvelle signification lorsqu'ils portent (en partie ou en totalité) sur des objets de type classe.

Le mécanisme de la surdéfinition d'opérateurs

Pour surdéfinir un opérateur existant `op`, on définit une fonction nommée `operator op` (on peut placer un ou plusieurs espaces entre le mot `operator` et l'opérateur, mais ce n'est pas une obligation) :

- soit sous forme d'une fonction indépendante (généralement amie d'une ou de plusieurs classes) ;
- soit sous forme d'une fonction membre d'une classe.

Dans le premier cas, si `op` est un opérateur binaire, la notation `a op b` est équivalente à :

```
operator op (a, b)
```

Dans le second cas, la même notation est équivalente à :

```
a.operator op (b)
```

Les possibilités et les limites de la surdéfinition d'opérateurs

On doit se limiter aux opérateurs existants, en conservant leur « pluralité » (unaire, binaire). Les opérateurs ainsi surdéfinis gardent leur priorité et leur associativité habituelle (voir tableau récapitulatif, un peu plus loin).

Un opérateur surdéfini doit toujours posséder un opérande de type classe (on ne peut donc pas modifier les significations des opérateurs usuels). Il doit donc s'agir :

- soit d'une fonction membre, auquel cas elle dispose obligatoirement d'un argument implicite du type de sa classe (`this`) ;
- soit d'une fonction indépendante (ou plutôt amie) possédant au moins un argument de type classe.

Il ne faut pas faire d'hypothèse sur la signification a priori d'un opérateur ; par exemple, la signification de `+=` pour une classe ne peut en aucun cas être déduite de la significaiton de `+` et de `=` pour cette même classe.

Cas particuliers

Les opérateurs `[]`, `()`, `->`, `new` et `delete` doivent obligatoirement être définis comme fonctions membre.

Les opérateurs `=` (affectation) et `&` (pointeur sur) possèdent une signification prédéfinie pour les objets de n'importe quel type classe. Cela ne les empêche nullement d'être surdéfinis. En ce qui concerne l'opérateur d'affectation, on peut choisir de transmettre son unique argument par valeur ou par référence. Dans le dernier cas, on ne perdra pas de vue que le seul moyen d'autoriser l'affectation d'une expression consiste à déclarer cet argument constant.

La surdéfinition de `new`, pour un type classe donné, se fait par une fonction de prototype :

```
void * new (size_t)
```

Elle reçoit, en unique argument, la taille de l'objet à allouer (cet argument sera généré automatiquement par le compilateur, lors d'un appel de `new`), et elle doit fournir en retour l'adresse de l'objet alloué.

La surdéfinition de `delete`, pour un `type` donné, se fait par une fonction de prototype :

```
void delete (type *)
```

Elle reçoit, en unique argument, l'adresse de l'objet à libérer.

Tableau récapitulatif

Les opérateurs surdéfinissables en C++ (classés par priorité décroissante)

Pluralité	Opérateurs	Associativité
Binaire	`()`(3) `[]`(3) `->`(3)	->
Unaire	`+ - ++`(5) `--`(5) `! ~ * &`(1) `new`(4)(6) `new[]`(4)(6) `delete`(4)(6) `delete[]`(4)(6) `(cast)`	<-
Binaire	`* / %`	->
Binaire	`+ -`	->
Binaire	`<< >>`	->
Binaire	`< <= > >=`	->
Binaire	`== !=`	->
Binaire	`&`	->
Binaire	`^`	->
Binaire	`||`	->
Binaire	`&&`	->
Binaire	`|`	->
Binaire	`=`(1)(3) `+= -= *= /= %= &= ^= |= <<= >>=`	<-
Binaire	`,`	->

(1) S'il n'est pas surdéfini, il possède une signification par défaut.

(3) Doit être défini comme fonction membre.

(4) Soit à un « niveau global » (fonction indépendante), soit pour une classe (fonction membre).

(5) Lorsqu'ils sont définis de façon unaire, ces opérateurs correspondent à la notation « pré » ; mais il en existe une définition binaire (avec deuxième opérande fictif de type `int`) qui correspond à la notation « post ».

(6) On distingue bien `new` de `new[]` et `delete` de `delete[]`

Même lorsqu'on a surdéfini les opérateurs `new` et `delete` pour une classe, il reste possible de faire appel aux opérateurs `new` et `delete` usuels, en utilisant l'opérateur de résolution de portée (`::`).

Exercice 86

Énoncé

Soit une classe `vecteur3d` définie comme suit :

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
} ;
```

Définir les opérateurs `==` et `!=` de manière qu'ils permettent de tester la coïncidence ou la non-coïncidence de deux points :

a. en utilisant des fonctions membre ;

b. en utilisant des fonctions amies.

Solutions

a. Avec des fonctions membre

Il suffit donc de prévoir, dans la classe `vecteur3d`, deux fonctions membre de nom `operator ==` et `operator !=`, recevant un argument de type `vecteur3d` correspondant au second argument des opérateurs (le premier opérande étant fourni par l'argument implicite `this` des fonctions membre). Voici la déclaration complète de notre classe, accompagnée des définitions des opérateurs :

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
   int operator == (vecteur3d) ;
   int operator != (vecteur3d) ;
} ;

int vecteur3d::operator == (vecteur3d v)
{  if ( (v.x == x) && (v.y == y) && (v.z ==z) ) return 1 ;
            else return 0 ;
}
int vecteur3d::operator != (vecteur3d v)
{  return ! ( (*this) == v ) ;
}
```

Notez que, dans la définition de `!=`, nous nous sommes servis de l'opérateur `==`. En pratique, on sera souvent amené à réécrire entièrement la définition d'un tel opérateur, pour de simples raisons d'efficacité (d'ailleurs, pour les mêmes raisons, on placera « en ligne » les fonctions `operator ==` et `operator !=`).

b. Avec des fonctions amies

Il faut donc prévoir de déclarer comme amies, dans la classe `vecteur3d`, deux fonctions (`operator ==` et `operator !=`) recevant deux arguments de type `vecteur3d` correspondant aux deux opérandes des opérateurs Voici la nouvelle déclaration de notre classe, accompagnée des définitions des opérateurs :

```
class vecteur3d
```

```
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
   friend int operator == (vecteur3d, vecteur3d) ;
   friend int operator != (vecteur3d, vecteur3d) ;
} ;
int operator == (vecteur3d v, vecteur3d w)
{  if ( (v.x == w.x) && (v.y == w.y) && (v.z == w.z) ) return
1 ;
                                                    else return
0 ;
}
int operator != (vecteur3d v, vecteur3d w)
{  return ! ( v == w ) ;
}
```

Voici, à titre indicatif, un exemple de programme, accompagné du résultat fourni par son exécution, utilisant n'importe laquelle des deux classes `vecteur3d` que nous venons de définir :

```
#include "vecteur3d.h"
#include <iostream>
using namespace std ;
main()
{  vecteur3d v1 (3,4,5), v2 (4,5,6), v3 (3,4,5) ;
    cout << "v1==v2 : " << (v1==v2) << " v1!=v2 : " <<
(v1!=v2) << "\n" ;
    cout << "v1==v3 : " << (v1==v3) << " v1!=v3 : " <<
(v1!=v3) << "\n" ;
}

v1==v2 : 0 v1!=v2 : 1
v1==v3 : 1 v1!=v3 : 0
```

Exercice 87

Énoncé

Soit la classe `vecteur3d` ainsi définie :

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
} ;
```

Définir l'opérateur binaire `+` pour qu'il fournisse la somme de deux vecteurs, et l'opérateur binaire `*` pour qu'il fournisse le produit scalaire de deux vecteurs. On choisira ici des fonctions amies.

Solution

Il suffit de créer deux fonctions amies nommées `operator +` et `operator *`. Elles recevront deux arguments de type `vecteur3d` ; la première fournira en retour un objet de type `vecteur3d`, la seconde fournira en retour un `float`.

```
class vecteur3d
{  float x, y, z ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { x = c1 ; y = c2 ; z = c3 ;
     }
          friend   vecteur3d   operator       +   (vecteur3d,
vecteur3d) ;
   friend float operator * (vecteur3d, vecteur3d) ;
} ;
vecteur3d operator + (vecteur3d v, vecteur3d w)
{  vecteur3d res ;
   res.x = v.x + w.x ;
   res.y = v.y + w.y ;
   return res ;
}
float operator * (vecteur3d v, vecteur3d w)
{  return (v.x * w.x + v.y * w.y + v.z * w.z) ;
}
```

Remarque

Il est possible de transmettre par référence les arguments des deux fonctions amies concernées.

Souvent, dans ce cas, on utilisera le qualificatif `const` puisque la fonction est supposée ne pas modifier les valeurs de ses arguments. Par exemple :

```
vecteur3d operator + (const vecteur3d & v, const vecteur3d & w)
```

Dans ce cas, la fonction peut certes être appelée avec des arguments effectifs constants. Mais il ne faudra pas perdre de vue qu'elle peut aussi être appelée avec arguments fournis sous forme d'expressions de type `vecteur3d`, voire avec des arguments d'un type autre que `vecteur3d`, pour peu qu'il existe une conversion implicite appropriée (voir le chapitre relatif aux conversions définies par l'utilisateur).

En revanche, il n'est pas possible de demander à `operator +` de transmettre son résultat par référence, puisque ce dernier est créé dans la fonction même, sous forme d'un objet de classe automatique. En toute rigueur, `operator *` pourrait transmettre son résultat (`float`) par référence, mais cela n'a guère d'intérêt en pratique.

Exercice 88

Énoncé

Soit la classe `vecteur3d` ainsi définie :

```
class vecteur3d
{
   float v[3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     {   // à compléter
     }
} ;
```

Compléter la définition du constructeur (en ligne), puis définir l'opérateur `[]` pour qu'il permette d'accéder à l'une des trois composantes d'un vecteur, et cela aussi bien au sein d'une expression (`... = v1[i]`) qu'à gauche d'un opérateur d'affectation `(v1[i] = ...)` ; de plus, on cherchera à se protéger contre d'éventuels risques de débordement d'indice.

La définition du constructeur ne pose aucun problème. En ce qui concerne l'opérateur `[]`, C++ ne permet de le surdéfinir que sous la forme d'une fonction membre (cette exception est justifiée par le fait que l'objet concerné ne doit pas risquer d'être soumis à une conversion, lorsqu'il apparaît à gauche d'une affectation). Elle possédera donc un seul argument de type `int` et elle renverra un résultat de type `vecteur3d`.

Pour que notre opérateur puisse être utilisé à gauche d'une affectation, il faut absolument que le résultat soit renvoyé par référence. Pour nous protéger d'un éventuel débordement d'indice, nous avons simplement prévu que toute tentative d'accès à un élément en dehors des limites conduirait à accéder au premier élément.

Voici la déclaration de notre classe, accompagnée de la définition de la fonction `operator []` :

```
class vecteur3d
{
   float v [3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { v[0] = c1 ; v[1] = c2 ; v[2] = c3 ;
     }
   float & operator [] ( int) ;
} ;

float & vecteur3d::operator [] (int i)
{   if ( (i<0) || (i>2) ) i = 0 ;       // pour éviter un
"débordement"
   return v[i] ;
}
```

Remarque

À titre indicatif, voici un petit exemple de programme faisant appel à notre classe `vecteur3d`, accompagné du résultat de son exécution :

```
#include "vecteur3d.h"
#include <iostream>
using namespace std ;

main()
{
  vecteur3d v1 (3,4,5) ;
   int i ;
   cout << "v1 = " ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
   for (i=0 ; i<3 ; i++) v1[i] = i ;
   cout << "\nv1 = "  ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
}

v1 = 3 4 5
v1 = 0 1 2
```

Exercice 89

Énoncé

L'exercice 79 vous avait proposé de créer une classe `set_int` permettant de représenter des ensembles de nombres entiers :

```
class set_int
{
   int * adval ;        // adresse du tableau des valeurs
   int nmax ;         // nombre maxi d'éléments
   int nelem ;         // nombre courant d'éléments
 public :
   set_int (int = 20) ;     // constructeur
   ~set_int () ;         // destructeur
   ......           // autres fonctions membre
} ;
```

Son implémentation prévoyait de placer les différents éléments dans un tableau alloué dynamiquement ; aussi l'affectation entre objets de type `set_int` posait-elle des problèmes, puisqu'elle aboutissait à des objets différents comportant des pointeurs sur un même emplacement dynamique.

Modifier la classe `set_int` pour qu'elle ne présente plus de telles lacunes. On prévoira que tout objet de type `set_int` comporte son propre emplacement dynamique, comme on l'avait fait pour permettre la transmission par valeur. De plus, on s'arrangera pour que l'affectation multiple soit utilisable.

Solution

Nous sommes en présence d'un problème voisin de celui posé par le constructeur par recopie. Nous l'avions résolu en prévoyant ce que l'on appelle une « copie profonde » de l'objet concerné (c'est-à-dire une copie non seulement de l'objet lui-même, mais de toutes ses parties dynamiques). Quelques différences supplémentaires surgissent néanmoins. En effet, ici :

- on peut se trouver en présence d'une affectation d'un objet à lui-même ;
- avant affectation, il existe deux objets « complets » (avec leur partie dynamique), alors que dans le cas du constructeur par recopie, il n'existait qu'un seul objet, le second étant à créer.

Voici comment traiter l'affectation `b = a`, dans le cas où `b` est différent de `a` :

- libération de l'emplacement pointé par `b` ;
- création dynamique d'un nouvel emplacement dans lequel on recopie les valeurs de l'emplacement désigné par `a` ;
- mise en place des valeurs des membres donnée de `b`.

Si `a` et `b` désignent le même objet (on s'en assurera dans l'opérateur d'affectation, en comparant les adresses des objets concernés), on évitera d'appliquer ce mécanisme qui conduirait à un emplacement dynamique pointé par « personne », et qui, donc, ne pourrait jamais être libéré. En fait, on se contentera de... ne rien faire !

Par ailleurs, pour que l'affectation multiple (telle que `c = b = a`) fonctionne correctement, il est nécessaire que notre opérateur renvoie une valeur de

retour (elle sera de type `set_int`) représentant la valeur de son premier opérande (après affectation, c'est-à-dire la valeur de `b` après `b = a`).

Voici ce que pourrait être la définition de notre fonction `operator =` (en ce qui concerne la déclaration de la classe `set_int`, il suffirait d'y ajouter `set_int & operator =` *(*`set_int &`*)* :

```
set_int & set_int::operator = (set_int & e)   // ou :   const
set_int & e
//  surdéfinition  de  l'affectation  -  les  commentaires
correspondent à b = a
{
   if (this != &e)                    // on ne fait rien pour a
= a
      { delete [] adval ;               // libération partie
dynamique de b
        adval = new int [nmax = e.nmax] ; // allocation nouvel
ensemble
                                          // pour a
        nelem = e.nelem ;                   // dans lequel on
recopie
        int i ;                               // entièrement
l'ensemble b
        for (i=0 ; i<nelem ; i++)           // avec sa partie
dynamique
     adval[i] = e.adval[i] ;
     }
   return * this ;
}
```

Remarques

1. On associera obligatoirement `const` à la transmission par référence, si l'on souhaite pouvoir appliquer l'affectation à une expression de type `set_int`. Cela n'est pas nécessairement justifié ici.
2. Une telle surdéfinition d'un opérateur d'affectation pour une classe possédant des parties dynamiques ne sera valable que si elle est associée à une surdéfinition comparable du constructeur par recopie (pour la classe `set_int`, celle proposée dans la solution de l'exercice 26 convient parfaitement).
3. A priori, seule la valeur du premier opérande a vraiment besoin d'être transmise par référence (pour que `=` puisse le modifier !) ; cette

condition est obligatoirement remplie puisque notre opérateur doit être surdéfini comme fonction membre. Toutefois, en pratique, on utilisera également la transmission par référence, à la fois pour le second opérande et pour la valeur de retour, de façon à être plus efficace (en temps). Notez d'ailleurs que si l'opérateur = renvoyait son résultat par valeur, il y aurait alors appel du constructeur de recopie (la remarque précédente s'appliquerait alors à une simple affectation).

Exercice 90

Énoncé

Considérer à nouveau la classe `set_int` créée dans l'exercice 79 (et sur laquelle est également fondé l'exercice précédent) :

```
class set_int
{   int * adval ;                  // adresse du tableau des
valeurs
    int nmax ;                  // nombre maxi d'éléments
    int nelem ;                 // nombre courant d'éléments
 public :
    set_int (int = 20) ;        // constructeur
    ~set_int () ;               // destructeur
    ......
} ;
```

Adapter cette classe, de manière que :

- l'on puisse ajouter un élément à un ensemble de type `set_int` par (`e` désignant un objet de type `set_int` et `n` un entier) : `e < n` ;
- `e[n]` prenne la valeur `1` si `n` appartient à `e` et la valeur `0` dans le cas contraire. On s'arrangera pour qu'une instruction de la forme `e[i] = ...` soit **rejetée à la compilation**.

Solution

Il nous faut donc surdéfinir l'opérateur binaire <, de façon qu'il reçoive

comme opérandes un objet de type `set_int` et un entier. Bien que l'énoncé ne prévoie rien, il est probable que l'on souhaitera pouvoir écrire des choses telles que (`e` étant de type `set_int`, `n` et `p` des entiers) :

```
e < n < p ;
```

Cela signifie donc que notre opérateur devra fournir comme valeur de retour l'ensemble concerné, après qu'on lui aura ajouté l'élément voulu.

Nous pouvons ici utiliser indifféremment une fonction membre ou une fonction amie. Nous choisirons la première possibilité. Par ailleurs, nous transmettrons les opérandes et la valeur de retour par référence (c'est possible ici car l'objet correspondant n'est pas créé au sein de l'opérateur même, c'est-à-dire qu'il n'est pas de classe automatique) ; ainsi, notre opérateur fonctionnera même si le constructeur par recopie n'a pas été surdéfini (en pratique toutefois, il faudra le faire dès qu'on souhaitera pouvoir transmettre la valeur d'un objet de type `set_int` en argument).

En ce qui concerne l'opérateur `[]`, il doit être surdéfini comme fonction membre, comme l'impose le C++, et cela bien qu'ici une affectation telle `e[i] =` soit interdite (alors que c'est précisément pour l'autoriser que C++ oblige d'en faire une fonction membre !). Pour interdire de telles affectations, la démarche la plus simple consiste à faire en sorte que le résultat fourni par l'opérateur ne soit pas une `lvalue`, en la **transmettant par valeur**.

Voici ce que pourraient être la définition et la déclaration de notre nouvelle classe munie de ces deux opérateurs (notez que nous utilisons `[]` pour définir `<`) :

```
class set_int
{   int * adval ;                    // adresse du tableau des
valeurs
   int nmax ;                   // nombre maxi d'éléments
   int nelem ;                  // nombre courant d'éléments
 public :
   set_int (int = 20) ;         // constructeur
   ~set_int () ;                // destructeur
   set_int & operator < (int) ;
    int operator [] (int) ;        // attention résultat par
valeur
} ;

set_int::set_int (int dim)
```

```
{ adval = new int [nmax=dim] ;
  nelem = 0 ;
}

set_int::~set_int ()
{ delete [] adval ;
}

      /* surdéfinition de < */
set_int & set_int::operator < (int nb)
{   // on examine si nb appartient déjà à l'ensemble
    //   en utilisant l'opérateur []
   if ( ! (*this)[nb] && (nelem < nmax) )  adval [nelem++] =
nb ;
   return (*this) ;
}
       /* surdéfinition de [] */
int set_int::operator [] (int nb)      // attention résultat
par valeur
{  int i=0 ;
  // on examine si nb appartient déjà à l'ensemble
  //  (dans ce cas i vaudra nelem en fin de boucle)
   while ( (i<nelem) && (adval[i] != nb) ) i++ ;
   return (i<nelem) ;
}
```

À titre indicatif, voici un exemple d'utilisation accompagné du résultat de son exécution :

```
#include "set_int.h"
#include <iostream>
using namespace std ;
main()
{ set_int ens(10) ;
  ens < 25 < 2 < 25 < 3 ;
  cout << (ens[25]) << " " << (ens[5]) << "\n" ;
}
```

```
 1 0
```

Exercice 91

Énoncé

Soit une classe `vecteur3d` définie comme suit :

```
class vecteur3d
{  float v [3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { v[0] = c1 ; v[1] = c2 ; v[2] = c3 ;
     }
   //   à compléter
} ;
```

Définir l'opérateur `[]` de manière que :

- il permette d'accéder « normalement » à un élément d'un objet non constant de type `vecteur3d`, et cela aussi bien dans une expression qu'en opérande de gauche d'une affectation ;
- il ne permette que la consultation (et non la modification) d'un objet constant de type `vecteur3d` (autrement dit, si `v` est un tel objet, une instruction de la forme `v[i] = ...` devra être rejetée à la compilation).

Solution

Rappelons que lorsque l'on définit des objets constants (qualificatif `const`), il n'est pas possible de leur appliquer une fonction membre publique, sauf si cette dernière a été déclarée avec le qualificatif `const` (auquel cas, elle peut indifféremment être utilisée avec des objets constants ou non constants). Ici, nous devons donc définir une fonction membre constante de nom

```
operator [].
```

Par ailleurs, pour qu'une affectation de la forme `v[i] = ...` soit interdite, il est nécessaire que notre opérateur renvoie son résultat par valeur (et non par adresse comme on a généralement l'habitude de le faire).

Dans ces conditions, on voit qu'il est nécessaire de prévoir deux fonctions membre différentes, pour traiter chacun des deux cas : objet constant ou objet non constant. Le choix de la « bonne fonction » sera assuré par le compilateur, selon la présence ou l'absence de l'attribut `const` pour l'objet

concerné.

Voici la définition complète de notre classe, accompagnée de la définition des deux fonctions `operator []` :

```
class vecteur3d
{  float v [3] ;
  public :
   vecteur3d (float c1=0.0, float c2=0.0, float c3=0.0)
     { v[0] = c1 ; v[1] = c2 ; v[2] = c3 ;
     }
    float    operator [] (int) const ; // [] pour un vecteur
constant
   float & operator [] (int) ;       // [] pour un vecteur non
constant
} ;


            /******** operator [] pour un objet non constant
*******/
float & vecteur3d::operator [] (int i)
{   if ( (i<0) || (i>2) ) i = 0 ;       // pour éviter un
débordement
   return v[i] ;
}
             /********** operator [] pour un objet constant
*********/
float   vecteur3d::operator [] (int i) const
{   if ( (i<0) || (i>2) ) i = 0 ;       // pour éviter un
débordement
   return v[i] ;
}
```

À titre indicatif, voici un petit programme utilisant la classe `vecteur3d` ainsi définie, accompagné du résultat produit par son exécution :

```
#include "vecteur3d.h"
#include <iostream>   // voir N.B. du paragraphe Nouvelles
possibilités
                     // d'entrées-sorties du chapitre 2
using namespace std ;
main()
{  int i ;
   vecteur3d v1 (3,4,5) ;
   const vecteur3d v2 (1,2,3) ;
   cout << "V1 : " ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
   cout << "\nV2 : " ;
```

```
   for (i=0 ; i<3 ; i++) cout << v2[i] << " " ;
   for (i=0 ; i<3 ; i++) v1[i] = i ;
   cout << "\nV1 : " ;
   for (i=0 ; i<3 ; i++) cout << v1[i] << " " ;
//      v2[1] = 3 ;  est bien rejeté à la compilation
}
```

```
V1 : 3 4 5
V2 : 1 2 3
V1 : 0 1 2
```

Exercice 92

Énoncé

Définir une classe `vect` permettant de représenter des « vecteurs dynamiques d'entiers », c'est-à-dire dont le nombre d'éléments peut ne pas être connu lors de la compilation. Plus précisément, on prévoira de déclarer de tels vecteurs par une instruction de la forme :

```
vect t(exp) ;
```

dans laquelle `exp` désigne une expression quelconque (de type `entier`).

On définira, de façon appropriée, l'opérateur `[]` de manière qu'il permette d'accéder à des éléments d'un objet d'un type `vect` comme on le ferait avec un tableau classique.

On ne cherchera pas à résoudre les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets de type `vect`. En revanche, on s'arrangera pour qu'aucun risque de « débordement » d'indice n'existe.

NB. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici. Il montrera également comment se protéger des débordements d'indice par une technique de gestion d'exceptions.

Solution

Les éléments d'un objet de type `vect` doivent obligatoirement être rangés en mémoire dynamique. L'emplacement correspondant sera donc alloué par le constructeur qui en recevra la taille en argument. Le destructeur devra donc, naturellement, libérer cet emplacement. En ce qui concerne l'accès à un élément, il se fera en surdéfinissant l'opérateur `[]`, comme nous l'avons déjà fait au cours des précédents exercices ; rappelons qu'il faudra obligatoirement le faire sous forme d'une fonction membre.

Pour nous protéger d'un éventuel débordement d'indice, nous ferons en sorte qu'une tentative d'accès à un élément situé en dehors du vecteur conduise à accéder à l'élément de rang `0`.

Voici ce que pourraient être la déclaration et la définition de notre classe :

```
   /********** déclaration de la classe vect ********/
class vect
{  int nelem ;        // nombre d'éléments
   int * adr ;         // adresse zone dynamique contenant les
éléments
 public :
   vect (int) ;              // constructeur
   ~vect () ;                // destructeur
    int & operator [] (int) ;  // accès à un élément par son
"indice"
} ;
   /*********** définition de la classe vect ********/
vect::vect (int n)
{  adr = new int [nelem = n] ;
   int i ;
   for (i=0 ; i<nelem ; i++) adr[i] = 0 ;
}
vect::~vect ()
{  delete [] adr ;
}
int & vect::operator [] (int i)
{  if ( (i<0)  || (i>=nelem) ) i=0 ;    // protection
   return adr [i] ;
}
```

Voici un petit exemple de programme d'utilisation d'une telle classe, accompagné du résultat produit par son exécution :

```
#include "vect.h"
```

```
#include <iostream>
using namespace std ;
main()
{  vect v(6) ;
   int i ;
   for (i=0 ; i<6 ; i++) v[i] = i ;
   for (i=0 ; i<6 ; i++) cout << v[i] << " " ;
}
```

```
 0 1 2 3 4 5
```

Exercice 93

Énoncé

En s'inspirant de l'exercice précédent, on souhaite créer une classe `int2d` permettant de représenter des tableaux dynamiques d'entiers à deux indices, c'est-à-dire dont les dimensions peuvent ne pas être connues lors de la compilation. Plus précisément, on prévoira de déclarer de tels tableaux par une déclaration de la forme :

```
int2d t(exp1, exp2) ;
```

dans laquelle `exp1` et `exp2` désignent une expression quelconque (de type entier).

On surdéfinira l'opérateur `()`, de manière qu'il permette d'accéder à des éléments d'un objet d'un type `int2d` comme on le ferait avec un tableau classique.

Là encore, on ne cherchera pas à résoudre les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets de type `int2d`. En revanche, on s'arrangera pour qu'il n'existe aucun risque de débordement d'indice.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici. Il montrera également comment se protéger contre les débordements d'indice par une technique de gestion d'exceptions.

Solution

Comme dans l'exercice précédent, les éléments d'un objet de type `int2d` doivent obligatoirement être rangés en mémoire dynamique. L'emplacement correspondant sera donc alloué par le constructeur qui en déduira la taille des deux dimensions reçues en arguments Le destructeur libérera cet emplacement.

Nous devons décider de la manière dont seront rangés les différents éléments en mémoire, à savoir « par ligne » ou « par colonne » (en toute rigueur, cette terminologie fait référence à la façon dont on a l'habitude de visualiser des tableaux à deux dimensions, ce que l'on nomme une ligne correspondant en fait à des éléments ayant même valeur du premier indice). Nous choisirons ici la première solution (c'est celle utilisée par C ou C++ pour les tableaux à deux dimensions). Ainsi, un élément repéré par les valeurs `i` et `j` des 2 indices sera situé à l'adresse (`adv` désignant l'adresse de l'emplacement dynamique alloué au tableau) :

```
adv + i * ncol + j
```

En ce qui concerne l'accès à un élément, il se fera en surdéfinissant l'opérateur `()`, d'une manière comparable à ce que nous avions fait pour `[]` ; là encore, C++ impose que `()` soit défini comme fonction membre.

Pour nous protéger d'un éventuel débordement d'indice, nous ferons en sorte qu'une valeur incorrecte d'un indice conduise à « faire comme si » on lui avait attribué la valeur `0`.

Voici ce que pourraient être la déclaration et la définition de notre classe :

```
   /******** déclaration de la classe int2d ********/
class int2d
{  int nlig ;      // nombre de "lignes"
   int ncol ;      // nombre de "colonnes"
    int * adv ;     // adresse emplacement dynamique contenant
les valeurs
  public :
   int2d (int nl, int nc) ;       // constructeur
   ~int2d () ;                    // destructeur
    int & operator () (int, int) ;  // accès à un élément, par
ses 2 "indices"
} ;
```

```
   /*********** définition du constructeur **********/
int2d::int2d (int nl, int nc)
{  nlig = nl ;
   ncol = nc ;
   adv = new int [nl*nc] ;
   int i ;
   for (i=0 ; i<nl*nc ; i++) adv[i] = 0 ;   // mise à zéro
}

   /*********** définition du destructeur **********/
int2d::~int2d ()
{  delete [] adv ;
}

   /********** définition de l'opérateur () *********/
int & int2d::operator () (int i, int j)
{  if ( (i<0) || (i>=nlig) ) i=0 ;  // protections sur premier
indice
   if ( (j<0) || (j>=ncol) ) j=0 ;  // protections sur second
indice
   return * (adv + i * ncol + j) ;
}
```

Voici un petit exemple d'utilisation de notre classe `int2d`, accompagné du résultat fourni par son exécution :

```
#include "int2d.h"
#include <iostream>
using namespace std ;
main()
{   int2d t1 (4,3) ;
    int i, j ;
    for (i=0 ; i<4 ; i++)
        for (j=0 ; j<3 ; j++)
            t1(i, j) = i+j ;
    for (i=0 ; i<4 ; i++)
      { for (j=0 ; j<3 ; j++)
             cout << t1 (i, j) << " " ;
        cout << "\n" ;
      }
}
```

```
 0 1 2
 1 2 3
 2 3 4
 3 4 5
```

Remarques

1. Dans la pratique, on sera amené à définir deux opérateurs `()` comme nous l'avions fait dans l'exercice précédent pour l'opérateur `[]` : l'un pour des objets non constants (celui défini ici), l'autre pour des objets constants (il sera prévu de manière à ne pas pouvoir modifier l'objet sur lequel il porte).
2. Généralement, par souci d'efficacité, les opérateurs tels que `()` seront définis « en ligne ».
3. On aurait pu songer à employer l'opérateur `[]` dont l'utilisation s'avère plus naturelle que celle de l'opérateur `()`. Cependant, `[]` ne peut être surdéfini que sous forme binaire. Il n'est donc pas possible de l'employer sous la forme `t[i, j]` pour accéder à un élément d'un tableau dynamique. On pourrait alors penser à l'utiliser sous la forme habituelle `t[i][j]`. Or, cette écriture doit être interprétée comme `(t[i]) [j])`, ce qui signifie qu'on n'y applique plus le second opérateur à un objet de type `int2d`.

   Comme, de surcroît, `[]` doit être défini comme fonction membre, l'écriture en question demanderait de définir `[]` pour une classe `int2d`, en s'arrangeant pour qu'il fournisse un résultat (« vecteur ligne ») lui-même de type classe. Dans ce dernier cas, il faudrait également surdéfinir l'opérateur `[]` pour qu'il fournisse un résultat de type `int`. Pour obtenir le résultat souhaité, il serait alors nécessaire de définir d'autres classes que `int2d`.

Exercice 94

Énoncé

Créer une classe nommée `histo` permettant de manipuler des « histogrammes ». On rappelle que l'on obtient un histogramme à partir d'un ensemble de valeurs `x(i)`, en définissant `n` tranches (intervalles) contiguës (souvent de même amplitude) et en comptabilisant le nombre de

valeurs `x(i)` appartenant à chacune de ces tranches.

On prévoira :

- un constructeur de la forme `histo (float min, float max, int ninter)`, dont les arguments précisent les bornes (`min` et `max`) des valeurs à prendre en compte et le nombre de tranches (`ninter`) supposées de même amplitude ;
- un opérateur `<<` défini tel que `h<<x` ajoute la valeur `x` à l'histogramme `h`, c'est-à-dire qu'elle incrémente de `1` le compteur relatif à la tranche à laquelle appartient `x`. Les valeurs sortant des limites (`min - max`) ne seront pas comptabilisées ;
- un opérateur `[]` défini tel que `h[i]` représente le nombre de valeurs répertoriées dans la tranche de rang `i` (la première tranche portant le numéro `1` ; un numéro incorrect de tranche conduira à considérer celle de rang `1`). On s'arrangera pour qu'une instruction de la forme `h[i] = ...` soit rejetée en compilation.

On ne cherchera pas ici à régler les problèmes posés par l'affectation ou la transmission par valeur d'objets du type `histo`.

Solution

Nous n'avons pas besoin de conserver les différentes valeurs `x(i)`, mais seulement les compteurs relatifs à chaque tranche. En revanche, il faut prévoir d'allouer dynamiquement (dans le constructeur) l'emplacement nécessaire à ces compteurs, puisque le nombre de tranches n'est pas connu lors de la compilation de la classe `histo`. Il faudra naturellement prévoir de libérer cet emplacement dans le destructeur de la classe. Les membres donnée de `histo` seront donc les valeurs extrêmes (minimale et maximale), le nombre de tranches et un pointeur sur l'emplacement contenant les compteurs.

L'opérateur `<<` peut être surdéfini, soit comme fonction membre, soit comme fonction amie ; nous choisirons ici la première solution. En revanche, l'opérateur `[]` doit absolument être surdéfini comme fonction membre. Pour qu'il ne soit pas possible d'écrire des affectations de la `forme h[i] = ...`, on fera en sorte que cet opérateur fournisse son résultat par valeur.

Voici ce que pourrait être la déclaration de notre classe `histo` :

```
   /*** fichier histo.h : déclaration de la classe histo ***/
class histo
{   float min ;  // borne inférieure
    float max ;  // borne supérieure
    int nint ;   // nombre de tranches utiles
     int * adc ;   // pointeur sur les compteurs associés à
chaque intervalle
                 // adc [i-1] = compteur valeurs de la tranche
de rang i
      float ecart ;// larg d'une tranche (pour éviter un
recalcul systématique)
  public :
    histo (float=0.0, float=1.0, int=10) ; // constructeur
    ~histo () ;                      // destructeur
    histo & operator << (float) ;   // ajoute une valeur
     int operator [] (int) ;          // nombre de valeurs dans
chaque tranche
} ;
```

Notez que nous avons prévu des valeurs par défaut pour les arguments du constructeur (celles-ci n'étaient pas imposées par l'énoncé).

En ce qui concerne la définition des différents membres, il faut noter qu'il est indispensable qu'une telle classe soit protégée contre toute utilisation incorrecte. Certes, cela passe par un contrôle de la valeur du numéro de tranche fourni à l'opérateur `[]`, ou par le refus de prendre en compte une valeur hors limites (qui fournirait un numéro de tranche conduisant à un emplacement situé en dehors de celui alloué par le constructeur).

Mais nous devons de surcroît nous assurer que les valeurs des arguments fournis au constructeur ne risquent pas de mettre en cause le fonctionnement ultérieur des différentes fonctions membre. En particulier, il est bon de vérifier que le nombre de tranches n'est pas négatif (notamment, une valeur nulle conduirait dans `<<` à une division par zéro) et que les valeurs du minimum et du maximum sont convenablement ordonnées et différentes l'une de l'autre (dans ce dernier cas, on aboutirait encore à une division par zéro). Ici, nous avons prévu de régler ce problème en attribuant le cas échéant des valeurs par défaut arbitraires (`max = min + 1, nint = 1`).

Voici ce que pourrait être la définition des différentes fonctions membre de notre classe `histo` :

```
                /************** définition de la classe histo
```

```
**********/
#include "histo.h"
#include <iostream>
using namespace std ;

/********************* constructeur ********************/
histo::histo (float mini, float maxi, int ninter)
{           // protection contre arguments erronés
    if (maxi < mini)
       { float temp = maxi ; maxi = mini ; mini = temp ; }
    if (maxi == mini) maxi = mini + 1.0 ;   // arbitraire
    if (ninter < 1) nint =1 ;
    min = mini ; max = maxi ; nint = ninter ;
      adc = new int [nint] ;                          // alloc
emplacements compteurs
    int i ;
    for (i=0 ; i<=nint-1 ; i++) adc[i] = 0 ; // et r.a.z.
     ecart = (max - min) / nint ;              // largeur d'une
tranche
}

                          /*********************    destructeur
**********************/
histo::~histo ()
{   delete [] adc ;
}

                        /*********************    opérateur    <<
********************/
histo & histo::operator << (float v)
{   int nt =  (v-min) / ecart ;
    // on ne comptabilise que les valeurs "convenables"
    if ( (nt>=0) && (nt<=nint-1) )  adc [nt] ++ ;
    return (*this) ;
}
                        /*********************    opérateur    []
********************/
int histo::operator [] (int n)          // résultat par valeur
ici
{     if  (  (n<1)  ||  (n>nint)  )  n=1  ;        //  protection
"débordement"
    return adc[n-1] ;
}
```

Voici, à titre indicatif, un petit programme d'essai de la classe `histo`, accompagné du résultat fourni par son exécution :

```
#include "histo.h"
#include <iostream>
using namespace std ;

main()
{   const int nint = 4 ;
    int i ;
    histo h (0., 5., nint) ;     // 4 tranches entre 0 et 5
     h << 1.5 << 2.4 << 3.8 << 3.0 << 2.0 << 3.5 << 2.8 <<
4.6 ;
    h << 12.0 << -3.5 ;
    for (i=0 ; i<10 ; i++) h << i/2.0 ;
    cout << "valeurs des tranches \n" ;
    for (i=1 ; i<=nint ; i++)
       cout << "numéro " << i << " : " << h[i] << "\n"  ;
    // une affectation telle que  h[2] = ... serait rejetée en
compilation
}
```

```
 valeurs des tranches
 numéro 1 : 3
 numéro 2 : 5
 numéro 3 : 6
 numéro 4 : 4
```

Exercice 95

Énoncé

Réaliser une classe nommée `stack_int` permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un emplacement alloué dynamiquement ; sa dimension sera déterminée par l'argument fourni à son constructeur (on lui prévoira une valeur par défaut de 20). Cette classe devra comporter les opérateurs suivants (nous supposons que `p` est un objet de type `stack_int` et `n` un entier) :

- `<<`, tel que `p<<n` ajoute l'entier `n` à la pile `p` (si la pile est pleine, rien ne se passe) ;

- `>>`, tel que `p>>n` place dans `n` la valeur du haut de la pile `p`, en la supprimant de la pile (si la pile est vide, la valeur de `n` ne sera pas modifiée) ;
- `++`, tel que `p++` vale `1` si la pile `p` est pleine et `0` dans le cas contraire ;
- `--`, tel que `p--` vale `1` si la pile `p` est vide et `0` dans le cas contraire.

On prévoira que les opérateurs `<<` et `>>` pourront être utilisés sous les formes suivantes (`n1, n2` et `n3` étant des entiers) :

```
p << n1 << n2 << n3 ;          p >> n1 >> n2 << n3 ;
```

On fera en sorte qu'il soit possible de transmettre une pile par valeur. En revanche, l'affectation entre piles ne sera pas permise, et on s'arrangera pour que cette situation aboutisse à un arrêt de l'exécution.

N.B. Le chapitre 17 vous montrera comment résoudre le présent exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

La classe `stack_int` contiendra comme membres donnée : la taille de l'emplacement réservé pour la pile (`nmax`), le nombre d'éléments placés à un moment donné sur la pile (`nelem`) et un pointeur sur l'emplacement qui sera alloué par le constructeur pour y ranger les éléments de la pile (`adv`). Notez qu'il n'est pas nécessaire de prévoir une donnée supplémentaire pour un éventuel « pointeur » de pile, dans la mesure où c'est le nombre d'éléments `nelem` qui joue ici ce rôle.

Les opérateurs requis peuvent indifféremment être définis comme fonctions membre ou comme fonctions amies. Nous choisirons ici la première solution. Pour que les opérateurs `<<` et `>>` puissent être utilisés à plusieurs reprises dans une même expression, il est nécessaire que, par exemple :

```
p << n1 << n2 << n3 ;
```

soit équivalent à :

```
( ( (p << n1) << n2 ) << n3 ) ;
```

Pour ce faire, les opérateurs `<<` et `>>` doivent fournir comme résultat la pile

reçue en premier opérande, après qu'elle a subi l'opération voulue (empilage ou dépilage). Il est préférable que ce résultat soit transmis par référence (on évitera la perte de temps due au constructeur par recopie).

En ce qui concerne la transmission par valeur d'un objet de type `stack_int`, nous ne pouvons pas nous contenter du constructeur par recopie par défaut puisque ce dernier ne recopierait pas la partie dynamique de l'objet, ce qui poserait les problèmes habituels. Nous devons donc surdéfinir le constructeur par recopie de façon qu'il réalise une « copie profonde » de l'objet.

Pour satisfaire à la contrainte imposée par l'énoncé sur l'affectation, nous allons surdéfinir l'opérateur d'affectation. Comme ce dernier doit se contenter d'afficher un message d'erreur et d'interrompre l'exécution, il n'est pas nécessaire qu'il renvoie une quelconque valeur (d'où la déclaration `void`).

Voici ce que pourrait être la déclaration de notre classe :

```
#include <stdlib.h>
#include <iostream>
using namespace std ;
class stack_int
{   int nmax ;                 // nombre maximal de la valeurs
de la pile
    int nelem ;                // nombre courant de valeurs de
la pile
    int * adv ;                // pointeur sur les valeurs
  public :
    stack_int (int = 20) ;            // constructeur
    ~stack_int () ;                   // destructeur
     stack_int (stack_int &) ;           // constructeur par
recopie
                                        // voir remarque 1 ci-
après
     void operator = (stack_int &) ;    // affectation – voir
remarque 2
    stack_int & operator << (int) ;   // opérateur d'empilage
    stack_int & operator >> (int &) ; // opérateur de dépilage
                                      // (attention int &)
     int operator ++ () ;              // opérateur de test
pile pleine
     int operator -- () ;              // opérateur de test
pile vide
} ;
```

Remarques

1. Il est nécessaire que l'argument de `stack_int` soit transmis par référence. On peut utiliser `const` ; dans ce cas, on pourra initialiser un objet avec un objet constant ou une expression d'un type susceptible d'être converti implicitement dans le type `stack_int` (ce qui est ici le cas des types entiers ou flottants ; voir le chapitre relatif aux conversions définies par l'utilisateur).
2. En revanche, il n'est pas nécessaire que l'argument de `operator=` soit transmis par référence. Si l'on souhaitait pouvoir affecter des objets constants, il faudrait ajouter le qualificatif `const` ; dans ce cas, on pourrait alors, du même coup, affecter des expressions d'un type convertible dans le type `stack_int`, c'est-à-dire ici d'un type entier ou flottant (voir le chapitre relatif aux conversions définies par l'utilisateur).
3. L'opérateur `>>` doit absolument recevoir son deuxième opérande par référence, puisqu'il doit pouvoir en modifier la valeur.

Voici la définition des fonctions membre de `stack_int` :

```
#include "stack-int.h"
stack_int::stack_int (int n)
{   nmax = n ;
    adv = new int [nmax] ;
    nelem = 0 ;
}
stack_int::~stack_int ()
{   delete [] adv ;
}

stack_int::stack_int (stack_int & p)
{   nmax = p.nmax ; nelem = p.nelem ;
    adv = new int [nmax] ;
    int i ;
    for (i=0 ; i<nelem ; i++)
       adv[i] = p.adv[i] ;
}
void stack_int::operator = (stack_int & p)
{    cout << "*** Tentative d'affectation entre piles - arrêt
exécution ***\n" ;
    exit (1) ;
}
```

```
stack_int & stack_int::operator << (int n)
{   if (nelem < nmax) adv[nelem++] = n ;
    return (*this) ;
}
stack_int & stack_int::operator >> (int & n)
{   if (nelem > 0) n = adv[--nelem] ;
    return (*this) ;
}
int stack_int::operator ++ ()
{   return  (nelem == nmax) ;
}
int stack_int::operator -- ()
{   return (nelem == 0) ;
}
```

À titre indicatif, voici un petit programme d'utilisation de notre classe, accompagné d'un exemple d'exécution :

```
  /************ programme d'essai de stack_int *********/
#include "stack_int.h"
#include <iostream>
using namespace std ;
main()
{   void fct (stack_int) ;
    stack_int pile (40) ;
     cout << "pleine : " << pile++ << " vide : " << pile-- <<
"\n" ;
    pile << 1 << 2 << 3 << 4 ;
    fct (pile) ;
    int n, p ;
    pile >> n >> p ;     // on dépile 2 valeurs
     cout << "haut de la pile au retour de fct : " << n << " "
<< p << "\n" ;
    stack_int pileb (25) ;
    pileb = pile ;       // tentative d'affectation
}
void fct (stack_int pl)
{   cout << "haut de la pile reçue par fct : " ;
    int  n, p ;
    pl >> n >> p ;  // on dépile 2 valeurs
    cout << n << " " << p << "\n" ;
    pl << 12 ;      // on en ajoute une
}
```

```
 pleine : 0   vide : 1
 haut de la pile reçue par fct : 4 3
```

```
haut de la pile au retour de fct : 4 3
*** Tentative d'affectation entre piles - arrêt exécution
***
```

Exercice 96

Énoncé

Adapter la classe `point` :

```
class point
{
  int x, y ;
    // .......
  public :
    point (int abs=0, int ord=0)                          //
constructeur
    // .....
} ;
```

de sorte qu'elle dispose de deux fonctions membre permettant de connaître, à tout instant, le nombre total d'objets de type `point`, ainsi que le nombre d'objets dynamiques (c'est-à-dire créés par `new`) de ce même type.

Solution

Ici, il n'est plus possible de se contenter de comptabiliser les appels au constructeur et au destructeur. Il faut, en outre, comptabiliser les appels à `new` et `delete`. La seule possibilité pour y parvenir consiste à surdéfinir ces deux opérateurs pour la classe `point`. Le nombre de points total et le nombre de points dynamiques seront conservés dans des membres statiques (n'existant qu'une seule fois pour l'ensemble des objets du type `point`). Quant aux fonctions membre fournissant les informations voulues, il est préférable d'en faire des fonctions membre statiques (déclarées, elles aussi, avec l'attribut `static`), ce qui permettra de les employer plus facilement que si on en avait fait des fonctions membre ordinaires (puisqu'on pourra les appeler sans avoir

à les faire porter sur un objet particulier).

Voici ce que pourrait être notre classe `point` (déclaration et définition) :

```
#include <cstddef>                // pour size_t
#include <iostream>
using namespace std ;
class point
{   static int npt ;              // nombre total de points
      static int nptd ;                   // nombre de points
dynamiques
    int x, y ;
  public :
    point (int abs=0, int ord=0)           // constructeur
        { x=abs ; y=ord ;
          npt++ ;
        }
    ~point ()                              // destructeur
        { npt-- ;
        }

    void * operator new (size_t sz)        // new surdéfini
    { nptd++ ;
       return ::new char[sz] ;                 // appelle new
prédéfini
    }
    void operator delete (void * dp)
     { nptd-- ;
       ::delete (dp) ;                       // appelle delete
prédéfini
     }
    static int npt_tot ()
     { return npt ;
     }
    static int npt_dyn ()
     { return nptd ;
     }
} ;

int point::npt = 0 ;         // initialisation des membres
statiques de point
int point::nptd = 0 ;
```

Notez que, dans la surdéfinition de `new` et `delete`, nous avons fait appel aux opérateurs prédéfinis (par emploi de `::`) pour ce qui concerne la gestion de la mémoire.

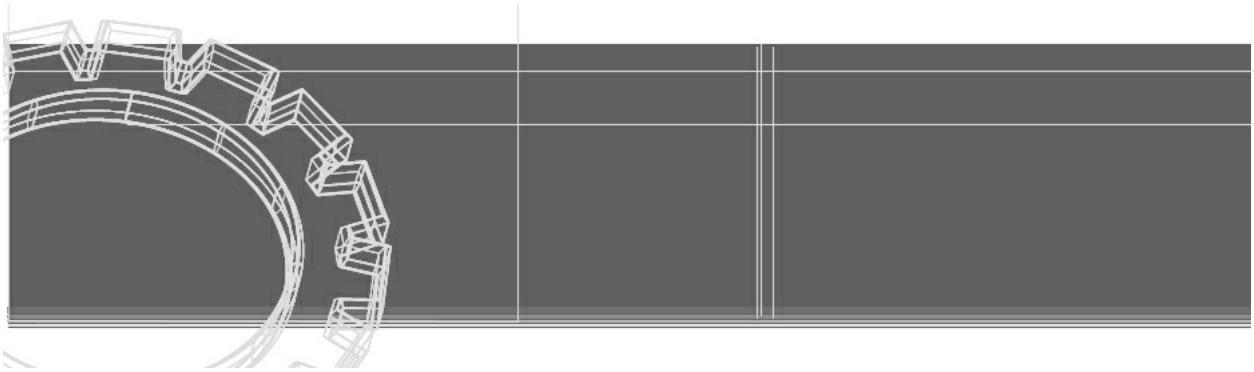
Voici un exemple de programme utilisant notre classe `point`, accompagné du

résultat fourni par son exécution :

```
#include "point.h"
#include <iostream>
using namespace std ;
main()
{
   point * ad1, * ad2 ;
   point a(3,5) ;
      cout << "A : " << point::npt_tot () << " " <<
point::npt_dyn () << "\n" ;
   ad1 = new point (1,3) ;
   point b ;
      cout << "B : " << point::npt_tot () << " " <<
point::npt_dyn () << "\n" ;
   ad2 = new point (2,0) ;
   delete ad1 ;
      cout << "C : " << point::npt_tot () << " " <<
point::npt_dyn () << "\n" ;
   point c(2) ;
   delete ad2 ;
       cout << "D : " << point::npt_tot () << " "
<< point::npt_dyn () << "\n" ;
}
```

```
A : 1 0
B : 3 1
C : 3 1
D : 3 0
```

Chapitre 12
Les conversions de type définies par l'utilisateur



Rappels

C++ vous permet de définir des conversions d'un type classe vers un autre type classe ou un type de base. On parle de conversions définies par l'utilisateur (en abrégé : C.D.U.). Ces conversions peuvent alors éventuellement être mises en œuvre de façon implicite par le compilateur, afin de donner une signification à un appel de fonction ou à un opérateur (sans conversion, l'appel ou l'opérateur serait illégal). On retrouve ainsi des possibilités comparables à celles qui nous sont offertes par le C en matière de conversions implicites.

Deux sortes de fonctions (obligatoirement des fonctions membre) permettent de définir des C.D.U. :

- les constructeurs à un argument (quel que soit le type de cet argument) réalisent une conversion du type de cet argument dans le type de sa

classe ; on peut cependant utiliser le mot-clé `explicit` devant la déclaration du constructeur pour en interdire l'utilisation dans une conversion implicite ;

- les opérateurs de `cast` ; dans une classe `A`, on définira un opérateur de conversion d'un type `x` (quelconque, c'est-à-dire aussi bien un type de base qu'un autre type classe) en introduisant la fonction membre de prototype :

```
operator x () ;
```

Notez que le type de la valeur de retour (obligatoirement défini par son nom) ne doit pas figurer dans l'en-tête ou le prototype d'une telle fonction.

Les règles d'utilisation des C.D.U. rejoignent celles concernant le choix d'une fonction surdéfinie.

- Les C.D.U. ne sont mises en œuvre que si cela est nécessaire.
- Une seule C.D.U. peut intervenir dans une chaîne de conversions (d'un argument d'une fonction ou d'un opérande d'un opérateur).
- Il ne doit pas y avoir d'ambiguïté, c'est-à-dire plusieurs chaînes de conversions conduisant au même type, pour un argument ou un opérande donné.

N.B. Aucune conversion n'est réalisable sur un argument effectif en cas de transmission par référence, sauf si l'argument muet correspondant est déclaré avec l'attribut `const` ; dans ce dernier cas, on retrouve les mêmes possibilités de conversion qu'en cas de transmission par valeur.

Exercice 97

Énoncé

Soit la classe `point` suivante :

```
class point
{    int x, y ;
   public :
     point (int abs=0, int ord=0)
```

```
        { x = abs ; y = ord ;
        }
    // .....
} ;
```

a. La munir d'un opérateur de `cast` permettant de convertir un `point` en un entier (correspondant à son abscisse).

b. Soient alors ces déclarations :

```
point p ;
int n ;
void fct (int) ;
```

Que font ces instructions :

```
n = p ;       //  instruction 1
fct (p) ;     //  instruction 2
```

Solutions

a. Il suffit de définir une fonction membre, de nom `operator int`, sans argument et renvoyant la valeur de l'abscisse du point l'ayant appelée. Rappelons que le type de la valeur de retour (que C++ déduit du nom de la fonction - ici `int`) ne doit pas figurer dans l'en-tête ou le prototype. Voici ce que pourraient être la déclaration et la définition de cette fonction, ici réunies en une seule déclaration d'une fonction en ligne :

```
operator int ()
{  return x ; }
```

b. L'instruction 1 est traduite par le compilateur en une conversion de `p` en `int` (par appel de `operator int`), suivie d'une affectation du résultat à `n`. Notez bien qu'il n'y a pas d'appel d'un quelconque opérateur d'affection de la classe `point`, ni d'un constructeur par recopie (car le seul argument transmis à la fonction `operator int` est l'argument implicite `this`).

L'instruction 2 est traduite par le compilateur en une conversion de `p` en `int` (par appel de `operator int`), suivie d'un appel de la fonction `fct`, à laquelle on transmet le résultat de cette conversion. Notez qu'il n'y pas, là non plus, d'appel d'un constructeur par recopie, ni pour `fct` (puisqu'elle reçoit un argument d'un type de base), ni pour `operator int` (pour la même raison que précédemment).

Exercice 98

Énoncé

Quels résultats fournira le programme suivant :

```
#include <iostream>
using namespace std ;
class point
{  int x, y ;
 public :
   point (int abs, int ord)              // constructeur 2
arguments
     { x = abs ; y = ord ;
     }
  operator int()                         // "cast" point -->
int
     { cout << "*** appel int() pour le point " << x << " "
<< y << "\n" ;
       return x ;
     }
} ;
main()
{  point a(1,5), b(2,8) ;
   int n1, n2, n3 ;
   n1 = a + 3 ;                          // instruction 1
   cout << "n1 = " << n1 << "\n" ;
   n2 = a + b ;                          // instruction 2
   cout << "n2 = " << n2 << "\n" ;
   double z1, z2 ;
   z1 = a + 3 ;                          // instruction 3
   cout << "z1 = " << z1 << "\n" ;
   z2 = a + b ;                          // instruction 4
   cout << "z2 = " << z2 << "\n" ;
}
```

Solution

Lorsque le compilateur rencontre, dans l'instruction 1, l'expression `a + 3`, il cherche tout d'abord s'il existe un opérateur surdéfini correspondant aux types `point` et `int` (dans cet ordre). Ici, il n'en trouve pas. Il va alors chercher à mettre en place des conversions permettant d'aboutir à une opération

existante ; ici, l'opérateur de `cast (operator int)` lui permet de se ramener à une addition d'entiers (par conversion de `p` en `int`), et c'est la seule possibilité. Le résultat est alors, bien sûr, de type `int` et il est affecté à `n1` sans conversion.

Le même raisonnement s'applique à l'instruction 2 ; l'évaluation de `a + b` se fait alors par conversion de `a` et de `b` en `int` (seule possibilité de donner une signification à l'opérateur +). Le résultat est de type `int` et il est affecté sans conversion à `n2`.

Les expressions figurant à droite des affectations des instructions 3 et 4 sont évaluées exactement suivant les mêmes règles que les expressions des instructions 1 et 2. Ce n'est qu'au moment de l'affectation du résultat (de type `int`) à la variable mentionnée que l'on opère une conversion `int --> double` (analogue à celles qui sont mises en place en langage C).

À titre indicatif, voici ce que fournit précisément l'exécution du programme :

```
*** appel int() pour le point 1 5
n1 = 4
*** appel int() pour le point 1 5
*** appel int() pour le point 2 8
n2 = 3
*** appel int() pour le point 1 5
z1 = 4
*** appel int() pour le point 1 5
*** appel int() pour le point 2 8
z2 = 3
```

Exercice 99

Énoncé

Quels résultats fournira le programme suivant :

```
#include <iostream>
using namespace std ;
class point
```

```
{
   int x, y ;
 public :
    point (int abs, int ord)             // constructeur 2
arguments
     { x = abs ; y = ord ;
     }
   operator int()                    // "cast" point --> int
      { cout << "** appel int() pour le point " << x << " "
<< y << "\n" ;
        return x ;
      }
} ;
void fct (double v)
{  cout << "$$ appel fct avec argument : " << v << "\n" ;
}

main()
{  point a(1,4) ;
   int n1 ;
   double z1, z2 ;
   n1 = a + 1.75 ;                   // instruction 1
   cout << "n1 = " << n1 << "\n" ;
   z1 = a + 1.75 ;                   // instruction 2
   cout << "z1 = " << z1 << "\n" ;
   z2 = a ;                          // instruction 3
   cout << "z2 = " << z2 << "\n" ;
   fct (a) ;                         // instruction 4
}
```

Pour évaluer l'expression `a + 1.75` de l'instruction 1, le compilateur met en place une chaîne de conversions de `a` en `double` (`point --> int` suivie de `int --> double`) de manière à aboutir à l'addition de deux valeurs de type `double` ; le résultat, de type `double`, est ensuite converti pour être affecté à `n1` (conversion forcée par l'affectation, comme d'habitude en C).

Notez bien qu'il n'est pas question pour le compilateur de prévoir la conversion en `int` de la valeur `1.75` (de façon à se ramener à l'addition de deux `int`, après conversion de `a` en `int`), car il s'agit là d'une « conversion dégradante » qui n'est jamais mise en œuvre de manière implicite dans un calcul d'expression. Il n'y a donc pas d'autre choix possible (notez que s'il y

en avait effectivement un autre, il ne s'agirait pas pour autant d'une situation d'ambiguïté dans la mesure où le compilateur appliquerait alors les règles habituelles de choix d'une fonction surdéfinie).

L'instruction 2 correspond à un raisonnement similaire, avec cette seule différence que le résultat de l'addition (de type `double`) peut être affecté à `z1` sans conversion.

Enfin les instructions 3 et 4 entraînent une conversion de `point` en `double`, par une suite de conversions `point --> int et int --> double`.

Voici le résultat de l'exécution du programme :

```
** appel int() pour le point 1 4
n1 = 2

** appel int() pour le point 1 4
z1 = 2.75

** appel int() pour le point 1 4
z2 = 1

** appel int() pour le point 1 4
$$ appel fct avec argument : 1
```

Exercice 100

Énoncé

Que se passerait-il si, dans la classe `point` du précédent exercice, nous avions introduit, en plus de l'opérateur `operator int`, un autre opérateur de `cast operator double` ?

Solution

Dans ce cas, les instructions 1 et 2 auraient conduit à une situation d'ambiguïté. En effet, le compilateur aurait disposé de deux chaînes de conversions permettant de convertir un point en `double` : soit `point --> double`, soit `point --> int` suivie de `int -> double`. En revanche, les instructions 3 et 4 auraient toujours été acceptées.

Exercice 101

Énoncé

Considérer la classe suivante :

```
class complexe
{  double x, y ;
  public :
   complexe (double r=0, double i=0) ;
    complexe (complexe &) ;          // ou complexe (const
complexe &)
} ;
```

Dans un programme contenant les déclarations :

```
complexe z (1,3) ;
void fct (complexe) ;
```

que produiront les instructions suivantes :

```
z = 3.75 ;          // instruction 1
fct (2.8) ;         // instruction 2
z = 2 ;             // instruction 3
fct (4) ;           // instruction 4
```

Solution

L'instruction 1 conduit à une conversion de la valeur `double 3.75` en un `complexe` par appel du constructeur à un argument (compte tenu des arguments par défaut), suivie d'une affectation à `z`.

L'instruction 2 conduit à une conversion de la valeur `double 2.8` en un `complexe` (comme précédemment par appel du constructeur à un argument) ; il y aura ensuite création d'une copie, par appel du constructeur par recopie de la classe `complexe`, copie qui sera transmise à la fonction `fct`.

Les instructions 3 et 4 jouent le même rôle que les deux précédentes, avec cette seule différence qu'elles font intervenir des conversions `int --> complexe` obtenues par une conversion `int --> double` suivie d'une conversion `double --> complexe`.

Exercice 102

Énoncé

a. Quels résultats fournira le programme suivant :

```
#include <iostream>
using namespace std ;
class point
{  int x, y ;
 public :
   point (int abs=0, int ord=0)        // constructeur 0, 1
ou 2 arguments
     { x = abs ; y = ord ;
       cout << "$$ construction point : " << x << " " << y
<< "\n" ;
     }
    friend point operator + (point, point) ;     // point +
point --> point
   void affiche ()
       { cout << "Coordonnées : " << x << " " << y <<
"\n" ;   }
} ;

point operator+ (point a, point b)
{  point r ;
   r.x = a.x + b.x ; r.y = a.y + b.y ;
   return r ;
}

main()
{  point a, b(2,4) ;
   a = b + 6 ;          // affectation 1
   a.affiche() ;
   a = 6 + b ;          // affectation 2
   b.affiche() ;
}
```

b. Même question en supposant que l'opérateur + a été surdéfini comme une fonction membre et non plus comme une fonction amie.

c. Même question en supposant que l'opérateur + est défini ainsi :

```
   friend point operator + (point &, point &) ;
```

d. Même question en supposant que l'opérateur + est défini ainsi :

```
friend point operator + (const point &, const point &) ;
```

Solutions

a. L'évaluation de l'expression `b + 6` de la première affectation se fait en utilisant l'opérateur + surdéfini pour la classe `point`, après avoir converti l'entier `6` en un `point` (par appel du constructeur à un argument). Le résultat, de type `point`, est alors affecté à `a`.

L'instruction d'affectation 2 se déroule de façon similaire (conversion de l'entier `6` en un `point`). En définitive, on obtient les résultats suivants :

```
$$ construction point : 0 0
$$ construction point : 2 4
$$ construction point : 6 0
$$ construction point : 0 0
Coordonnées : 8 4
$$ construction point : 6 0
$$ construction point : 0 0
Coordonnées : 2 4
```

b. En revanche, si l'opérateur + avait été surdéfini par une fonction membre, la seconde instruction d'affectation aurait été rejetée à la compilation ; en effet, elle aurait été interprétée comme :

```
.operator + (b)
```

On voit ici que seule la fonction amie permet de traiter de façon identique les deux opérandes d'un opérateur binaire, notamment en ce qui concerne les possibilités de conversions implicites.

c. La transmission par référence impose que l'argument effectif d'une fonction soit du même type que l'argument muet correspondant, aucune conversion n'étant alors possible. Les affectations 1 et 2 seront rejetées en compilation.

d. La présence de `const` permet au compilateur de transmettre à la fonction (`operator +`) un objet temporaire obtenu par d'éventuelles conversions des arguments effectifs. Toutes les affectations redeviennent correctes.

Exercice 103

Énoncé

Soient les deux classes suivantes :

```
class B
{    // ...
 public :
   B () ;        // constructeur sans argument
   B (int) ;     // constructeur à un argument entier
     // ...
} ;

class A
{    // ...
   friend operator + (A, A) ;
 public :
   A () ;        // constructeur sans argument
   A (int) ;     // constructeur à un argument entier
   A (B) ;       // constructeur à un argument de type B
     // ...
} ;
```

a. Dans un programme contenant les déclarations :

```
A a1, a2, a3 ;

B b1, b2, b3 ;
```

les instructions suivantes seront-elles correctes et, si oui, que feront-elles ?

```
a1 = b1 ;         // instruction 1
b1 = a1 ;         // instruction 2
a3 = a1 + a2 ;    // instruction 3
a3 = b1 + b2 ;    // instruction 4
b3 = a1 + a2 ;    // instruction 5
```

b. Comment obtenir le même résultat sans définir, dans `A`, le constructeur `A(B)` ?

Solutions

a.

Instruction 1

Il faut distinguer 2 cas :

Si `A` n'a pas surdéfini d'opérateur d'affectation d'un objet de type `B` à un objet de type `A`, il y aura conversion de `b1` dans le type de `A` (par appel du constructeur `A(B)`), suivie d'une affectation à `a1` (en utilisant soit l'opérateur d'affectation par défaut de `A`, soit éventuellement celui qui aurait pu y être surdéfini).

La forme usuelle de l'en-tête de l'opérateur `A` est `A & operator = (const & A)`, mais les références ne sont pas indispensables, pas plus que `const`. De même, la valeur de retour peut ne pas exister dès lors qu'on ne cherche pas à pas réaliser des affectations multiples.

En revanche, si `A` a surdéfini un opérateur d'affectation d'un objet de type `B` à un objet de type `A`, ce dernier sera utilisé pour réaliser l'instruction 1, et, dans ce cas, il n'y aura donc pas d'appel de constructeur de `A`, ni d'éventuel autre opérateur d'affectation. Ce comportement s'explique par le fait que les C.D.U. ne sont mises en œuvre que lorsque cela est nécessaire.

Instruction 2

Là encore, il faut distinguer les deux cas précédents. Si `B` n'a pas surdéfini d'opérateur d'affectation d'un objet de type `B` à un objet de type `A`, l'instruction 2 conduira à une erreur de compilation puisqu'il n'existera aucune possibilité de convertir un objet de type `A` en un objet de type `B`. En revanche, si l'opérateur d'affectation en question existe, il sera tout simplement utilisé.

Instruction 3

Elle ne pose aucun problème particulier.

Instruction 4

Ici, `b1` et `b2` seront convertis dans le type `A` en utilisant le constructeur `A(B)`

avant d'être transmis à l'opérateur + (de `A`). Le résultat, de type `A`, sera affecté à `a3`, en utilisant l'opérateur d'affectation de `A` – soit celui par défaut, soit celui éventuellement surdéfini.

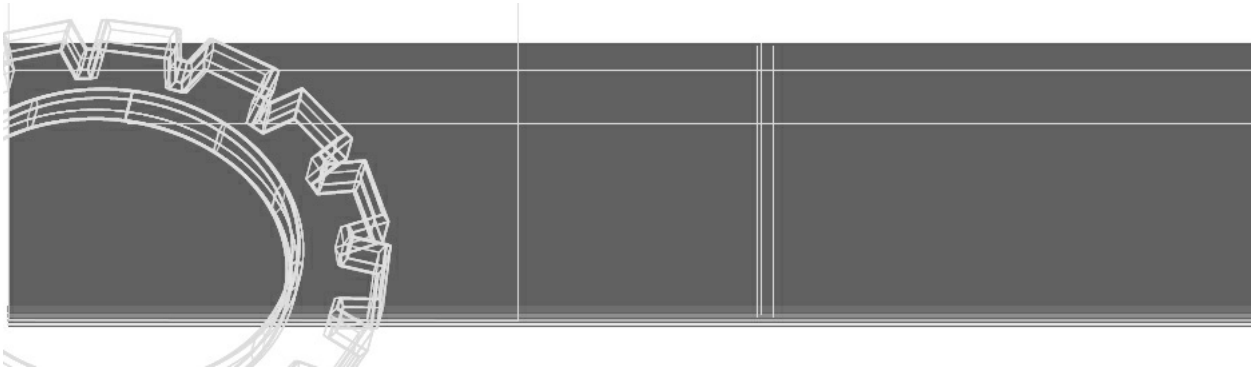
Instruction 5

L'expression `a1 + a2` ne pose pas de problème puisqu'elle est évaluée à l'aide de l'opérateur + de la classe `A` ; elle fournit un résultat de type `A`. En revanche, pour l'affectation du résultat à `b3`, il faut à nouveau distinguer les deux cas déjà évoqués. Si `B` a surdéfini un opérateur d'affectation d'un objet de type `B` à un objet de type `A`, il sera utilisé ; si un tel opérateur n'a pas été surdéfini, l'instruction sera rejetée par le compilateur (puisqu'il n'existera alors aucune possibilité de conversion de `B` en `A`).

b. En introduisant, dans la classe `B`, un opérateur de `cast` permettant la conversion de `B` en `A`, de la forme :

```
operator B ()
```

Chapitre 13
La technique de l'héritage



Rappels

Le concept d'héritage constitue l'un des fondements de la Programmation Orientée Objet. Il permet de définir une nouvelle classe *B* dite « dérivée », à partir d'une classe existante *A*, dite « de base » ; pour ce faire, on procède ainsi :

```
class B : public A    // ou :    private A     ou (depuis la
version 3)  protected A
{     //  définition des membres supplémentaires (données ou
fonctions)
      //  ou redéfinition de membres existants dans A (données
ou fonctions)
} ;
```

Avec `public A`, on parle de « dérivation publique » ; avec `private A`, on parle de « dérivation privée » ; avec `protected A`, on parle de « dérivation protégée ».

Modalités d'accès à la classe de base

Les membres privés d'une classe de base ne sont jamais accessibles aux fonctions membre de sa classe dérivée.

Outre les « statuts » public ou privé (présentés au chapitre 3), il existe un statut « protégé ». Un membre protégé se comporte comme un membre privé pour un utilisateur quelconque de la classe ou de la classe dérivée, mais comme un membre public pour la classe dérivée.

D'autre part, il existe trois sortes de dérivation :

- **publique** – les membres de la classe de base conservent leur statut dans la classe dérivée ; c'est la situation la plus usuelle ;
- **privée** – tous les membres de la classe de base deviennent privés dans la classe dérivée ;
- **protégée** (depuis la version 3) – les membres publics de la classe de base deviennent membres protégés de la classe dérivée ; les autres membres conservent leur statut.

Lorsqu'un membre (donnée ou fonction) est redéfini dans une classe dérivée, il reste toujours possible (soit dans les fonctions membre de cette classe, soit pour un client de cette classe) d'accéder aux membres de même nom de la classe de base ; il suffit pour cela d'utiliser l'opérateur de résolution de portée (`::`), sous réserve, bien sûr, qu'un tel accès soit autorisé.

Appel des constructeurs et des destructeurs

Soit `B` une classe dérivée d'une classe de base `A`. Naturellement, dès lors que `B` possède au moins un constructeur, la création d'un objet de type `B` implique obligatoirement l'appel d'un constructeur de `B`. Mais, de plus, ce constructeur de `B` doit prévoir des arguments à destination d'un constructeur de `A` (une exception a lieu soit si `A` n'a pas de constructeur, soit si `A` possède un constructeur sans argument). Ces arguments sont précisés dans la définition du constructeur de `B`, comme dans cet exemple :

```
B (int x, int y, char coul) : A (x, y) ;
```

Les arguments mentionnés pour `A` peuvent éventuellement l'être sous forme d'expressions.

Cas particulier du constructeur par recopie

En plus des règles ci-dessus, il faut ajouter que si la classe dérivée `B` ne possède pas de constructeur par recopie, il y aura appel du constructeur par recopie par défaut de `B`, lequel procédera ainsi :

- appel du constructeur par recopie de `A` (soit celui qui y a été défini, soit le constructeur par recopie par défaut) ;
- initialisation des membres donnée de `B` qui ne sont pas hérités de `A`.

En revanche, un problème se pose lorsque la classe dérivée définit explicitement un constructeur par recopie. En effet, dans ce cas, il faut tenir compte de ce que l'appel de ce constructeur par recopie entraînera l'appel :

- du constructeur de la classe de base mentionné dans son en-tête, comme dans cet exemple (il s'agit ici d'un constructeur par recopie de la classe de base, mais il pourrait s'agir de n'importe quel autre constructeur) :

```
  B (B & b) : A(b) ; // appel du constructeur par recopie de A
                       // auquel sera transmise la partie de B
héritée de A
                         // (grâce aux règles de compatibilité
entre
                     // classe dérivée et classe de base)
```

- d'un constructeur sans argument, si aucun constructeur de la classe de base n'est mentionné dans l'en-tête ; dans ce cas, il est nécessaire que la classe de base dispose d'un tel constructeur sans argument, faute de quoi on obtiendrait une erreur de compilation.

Conséquences de l'héritage

Considérons la situation suivante, dans laquelle la classe `A` possède une fonction membre `f` (dont nous ne précisons pas les arguments) fournissant un résultat de type `t` (quelconque : type de base ou type défini par l'utilisateur, éventuellement type classe) :

```
class A
{   .....
  public :
    t f (...) ;
    .....
} ;
```

```
class B : public A
{   .....
} ;
```

```
A a ;       // a est du type A
B b ;       // b est du type B, dérivé de A
```

Naturellement, un appel tel que `a.f(...)` a un sens et il fournit un résultat de type `t`. Le fait que `B` hérite publiquement de `A` permet alors de donner un sens à un appel tel que :

```
b.f (...)
```

La fonction `f` agira sur `b`, comme s'il était de type `A`. **Le résultat fourni par `f` sera cependant toujours de type `t`**, même, notamment, lorsque le type `t` est précisément le type `A` (le résultat de `f` pourra toutefois être soumis à d'éventuelles conversions s'il est affecté à une `lvalue`).

Cas particulier de l'opérateur d'affectation

Considérons une classe `B` dérivant d'une classe `A`.

Si la classe dérivée `B` n'a pas surdéfini l'opérateur d'affectation, l'affectation de deux objets de type `B` se déroule membre à membre, en considérant que la « partie héritée de `A` » constitue un membre. Ainsi, les membres propres à `B` sont traités par l'affectation prévue pour leur type (par défaut ou surdéfinie, suivant le cas). La partie héritée de `A` est traitée par l'affectation prévue dans la classe `A`.

Si la classe dérivée `B` a surdéfini l'opérateur =, l'affectation de deux objets de type `B` fera nécessairement appel à l'opérateur = défini dans `B`. Celui de `A` ne sera pas appelé, même s'il a été surdéfini. **Il faudra donc que l'opérateur = de `B` prenne en charge tout ce qui concerne l'affectation d'objets de type `B`**, y compris pour ce qui est des membres hérités de `A` (quitte à faire appel à l'opérateur d'affectation de `A`).

Compatibilité entre objets d'une classe de base et objets d'une classe dérivée

Considérons :

```
class A
{   .....
};
```

```
class B : public A
{   .....
 } ;
```

```
A a ;       // a est du type A
```

```
B b ;      // b est du type B, dérivé de A
A * ada ;  // ada est un pointeur sur des objets de type A
B * adb ;  // adb est un pointeur sur des objets de type B
```

Il existe deux conversions implicites :

- d'un objet d'un type dérivé dans un objet d'un type de base. Ainsi l'affectation `a = b` est légale : elle revient à convertir `b` dans le type `A` (c'est-à-dire, en fait, à ne considérer de `b` que ce qui est du type `A`) et à affecter ce résultat à `a` (avec appel, soit de l'opérateur d'affectation de `A` si celui-ci a été surdéfini, soit de l'opérateur d'affectation par défaut de `A`). L'affectation inverse `b = a` est, quant à elle, illégale ;
- d'un pointeur sur une classe dérivée en un pointeur sur une classe de base. Ainsi l'affectation `ada = adb` est légale, tandis que `adb = ada` est illégale (elle peut cependant être forcée par emploi de l'opérateur de `cast` : `adb = (B*) ada).`

Exercice 104

Énoncé

On dispose d'un fichier nommé `point.h` contenant la déclaration suivante de la classe `point` :

```
class point
{   float x, y ;
  public :
    void initialise (float abs=0.0, float ord=0.0)
       { x = abs ; y = ord ;
       }
    void affiche ()
       { cout << "Point de coordonnées : " << x << " " << y
<< "\n" ;
       }
    float abs ()  { return x ; }
    float ord ()  { return y ; }
} ;
```

a. Créer une classe `pointb`, dérivée de `point` comportant simplement une nouvelle fonction membre nommée `rho`, fournissant la valeur du rayon

vecteur (première coordonnée polaire) d'un point.

b. Même question, en supposant que les membres `x` et `y` ont été déclarés protégés (`protected`) dans `point`, et non plus privés.

c. Introduire un constructeur dans la classe `pointb`.

d. Quelles sont les fonctions membre utilisables pour un objet de type `pointb` ?

Solutions

a. Il suffit de prévoir, dans la déclaration de `pointb`, une nouvelle fonction membre de prototype :

```
float rho () ;
```

Toutefois, comme les membres `x` et `y` sont privés, ils restent privés pour les fonctions membre de sa classe dérivée `pointb` ; ce qui signifie qu'au sein de la définition de `rho`, il faut faire appel aux « fonctions d'accès » de `point` que sont `abs` et `ord`. Voici ce que pourrait être notre classe `pointb` (ici, nous avons fourni `rho` sous forme d'une fonction en ligne) :

```
#include "point.h"              // pour la déclaration de la
classe point
#include <math.h>
class pointb : public point
{ public :
    float rho ()
       { return  sqrt (abs () * abs () + ord () * ord () ) ;
       }
} ;
```

Notez que, telle qu'elle a été définie, la classe `point` n'a pas donné naissance à un fichier objet (puisque toutes ses fonctions membre sont en ligne). Il en va de même ici pour `pointb`. Aussi, pour utiliser `pointb` au sein d'un programme, il suffira d'inclure les fichiers contenant les déclarations de `pointb`. Naturellement, dans la pratique, il en ira rarement ainsi ; en général, on devra fournir non seulement les déclarations de la classe de base et de sa classe dérivée, mais également les fichiers objet correspondant à leurs compilations respectives.

b. La définition précédente reste valable mais, néanmoins, comme les

membres `x` et `y` de `point` ont été déclarés protégés, ils sont accessibles aux fonctions membre de sa classe dérivée ; aussi est-il possible, dans la définition de `rho`, de les utiliser « directement ». Voici ce que pourrait devenir notre fonction `rho` (toujours ici « en ligne ») :

```
    float rho ()                          // il faut que x et y
soient
          { return   sqrt (x * x + y * y) ;   // déclarés
"protected" dans point
       }
```

c. Voici ce que pourrait être un constructeur à deux arguments (avec valeurs par défaut) :

```
    pointb (float c1=0.0, float c2=0.0)
       { initialise (c1, c2) ;
       }
```

Là encore, si les membres `x` et `y` de *point* ont été déclarés « protégés », il est possible d'écrire ainsi notre constructeur :

```
    pointb (float c1=0.0, float c2=0.0)
        { x = c1 ; y = c2 ;             // il faut que x et y
soient
       }                                // déclarés "protected"
dans point
```

Notez qu'ici il n'est pas possible au constructeur de `pointb` d'appeler un quelconque constructeur de `point` puisque ce dernier type ne possède pas de constructeur.

d. Un objet de type `pointb` peut utiliser n'importe laquelle des fonctions membre publiques de `point`, c'est-à-dire `initialise`, `affiche`, `abs` et `ord`, ainsi que n'importe laquelle des fonctions membre publiques de `pointb`, c'est-à-dire `rho` ou le constructeur `pointb`. Notez d'ailleurs qu'ici le constructeur et `initialise` font double emploi : cela provient d'une part de ce que `point` ne dispose pas de véritable constructeur, d'autre part de ce que `pointb` n'a pas défini de membres donnée supplémentaires, de sorte qu'il n'y a rien de plus à faire pour initialiser un objet de type `pointb` que pour initialiser un objet de type `point`.

Exercice 105

Énoncé

On dispose d'un fichier `point.h` contenant la déclaration suivante de la classe `point` :

```
#include <iostream>
using namespace std ;
class point
{   float x, y ;
  public :
    point (float abs=0.0, float ord=0.0)
       { x = abs ; y = ord ;
       }
    void affiche ()
       { cout << "Coordonnées : " << x << " " << y << "\n" ;
       }
    void deplace (float dx, float dy)
       { x =  x + dx ;  y = y + dy ;
       }
} ;
```

a. Créer une classe `pointcol`, dérivée de `point`, comportant :

- un membre donnée supplémentaire `cl`, de type `int`, contenant la « couleur » d'un point ;
- les fonctions membre suivantes :

  `affiche (redéfinie), qui affiche les coordonnées et la couleur d'un objet de type pointcol ;`
  `colore (int couleur), qui permet de définir la couleur d'un objet de type pointcol ;`
- un constructeur permettant de définir la couleur et les coordonnées (on ne le définira pas en ligne).

b. Que fera alors précisément cette instruction :

```
pointcol (2.5, 3.25, 5) ;
```

Solutions

a. La fonction `colore` ne pose aucun problème particulier puisqu'elle agit uniquement sur un membre donnée propre à `pointcool`. En ce qui

concerne affiche, il est nécessaire qu'elle puisse afficher les valeurs des membres x et y, hérités de point. Comme ces membres sont privés (et non protégés), il n'est pas possible que la nouvelle méthode affiche de pointcol accède à eux directement. Elle doit donc obligatoirement faire appel à la méthode affiche du type point ; il suffit, pour cela, d'utiliser l'opérateur de résolution de portée. Enfin, le constructeur de pointcol doit retransmettre au constructeur de point les coordonnées qu'il aura reçues par ses deux premiers arguments.

Voici ce que pourrait être notre classe pointcol (ici, toutes les fonctions membre, sauf le constructeur, sont en ligne) :

```
          /****** fichier pointcol.h :déclaration de pointcol
******/
#include "point.h"
#include <iostream>
using namespace std ;
class pointcol : public point
{   int cl ;
  public :
    pointcol (float = 0.0, float = 0.0, int = 0) ;
    void colore (int coul)
      { cl = coul ;
      }
    void affiche ()                    // affiche doit appeler
affiche de
       { point::affiche () ;               // point pour les
coordonnées
        cout << "  couleur : " << cl ; // mais elle a accès à
la couleur
      }
} ;

         /****** définition du constructeur de pointcol *****/
#include "point.h"
#include "pointcol.h"
pointcol::pointcol (float abs, float ord, int coul) : point
(abs, ord)
{   cl = coul ;          // on pourrait aussi écrire colore
(coul) ;
}
```

Notez bien que l'on précise le constructeur de point devant être appelé par celui de pointcol, au niveau du constructeur de pointcol, et non de sa déclaration.

b. La déclaration `pointcol (2.5, 3.25, 5)` entraîne la création d'un emplacement pour un objet de type `pointcol`, lequel est initialisé par appel, successivement :

- du constructeur de `point`, qui reçoit en argument les valeurs `2.5` et `3.25` (comme prévu dans l'en-tête du constructeur de `pointcol`) ;
- du constructeur de `pointcol`.

Exercice 106

Énoncé

On suppose qu'on dispose de la même classe `point` (et donc du fichier `point.h`) que dans l'exercice précédent. Créer une classe `pointcol` possédant les mêmes caractéristiques que ci-dessus, mais sans faire appel à l'héritage. Quelles différences apparaîtront entre cette classe `pointcol` et celle de l'exercice précédent, au niveau des possibilités d'utilisation ?

Solution

La seule démarche possible consiste à créer une classe `pointcol` dans laquelle un des membres donnée est lui-même de type `point`. Sa déclaration et sa définition se présenteraient alors ainsi :

```
            /***** fichier pointcol.h : déclaration de pointcol
*****/
#include "point.h"
#include <iostream>
using namespace std ;
class pointcol
{   point p ;
    int cl ;
  public :
    pointcol (float = 0.0, float = 0.0, int = 0) ;
    void colore (int coul)
      { cl = coul ;
      }
    void affiche ()
```

```
        {   p.affiche () ;                      // affiche doit
appeler affiche
           cout  << "  couleur : " << cl ; // du point p pour
les
                                              // coordonnées
       }
} ;
         /****** définition du constructeur de pointcol *****/
#include "point.h"
#include "pointcol.h"
pointcol::pointcol (float abs, float ord, int coul) : p (abs,
ord)
{  cl = coul ;
}
```

Apparemment, il existe une analogie étroite entre cette classe `pointcol` et celle de l'exercice précédent. Néanmoins, l'utilisateur de cette nouvelle classe ne peut plus faire directement appel aux fonctions membre héritées de `point`. Ainsi, pour appliquer la méthode `deplace` à un objet `a` de type `point`, il devrait absolument écrire : `a.p.deplace (...)` ; or, cela n'est pas autorisé ici, puisque `p` est un membre privé de `pointcol`.

Exercice 107

Énoncé

Soit une classe `point` ainsi définie (nous ne fournissons pas la définition de son constructeur) :

```
class point
{   int x, y ;
  public :
    point (int = 0, int = 0) ;
    friend int operator == (point, point) ;
} ;

int operator == (point a, point b)
{   return  a.x == b.x && a.y == b.y ;
}
```

Soit la classe `pointcol`, dérivée de `point` :

```
class pointcol : public point
```

```
{   int cl ;
  public :
    pointcol (int = 0, int = 0, int = 0) ;
    //   éventuelles fonctions membre
}  ;
```

a. Si `a` et `b` sont de type `pointcol` et `p` de type `point`, les instructions suivantes sontelles correctes et, si oui, que font-elles ?

```
if (a == b) ...      // instruction 1
if (a == p) ...      // instruction 2
if (p == a) ...      // instruction 3
if (a == 5) ...      // instruction 4
if (5 == a) ...      // instruction 5
```

b. Mêmes questions, en supposant, cette fois, que l'opérateur `==` a été défini au sein de `point`, sous forme d'une fonction membre.

Solutions

a. Les 5 instructions proposées sont correctes. D'une manière générale, `x == y` est interprété comme `operator == (x, y)`. Si `x` et `y` sont de type `point`, aucun problème ne se pose bien sûr. Si l'un des opérandes est de type `pointcol` (ou les deux), il sera converti implicitement dans le type `point`. Si l'un des opérandes est de type `int`, il sera converti implicitement dans le type `point` (par utilisation du constructeur à un argument de `point`).

En ce qui concerne la signification de la comparaison, on voit qu'elle revient à ne considérer d'un objet de type `pointcol` que ses coordonnées. Pour un entier, elle revient à le considérer comme un `point` ayant cet entier pour abscisse et une ordonnée nulle.

N.B. Si les arguments de `operator=` étaient transmis par référence, les deux dernières affectations seraient rejetées, à moins d'avoir en plus prévu l'attribut `const`.

b. Cette fois, `x == y` est interprété comme `x.operator == (y)`. Si `x` est de type `point` et `y` d'un type pouvant se ramener au type `point` (c'est-à-dire soit du type `pointcol` qui sera converti implicitement en un type de base `point`, soit d'un type entier qui sera converti implicitement en un type `point` par l'intermédiaire du constructeur), aucun problème ne se pose (c'est le cas de la troisième instruction).

Si `x` est de type `pointcol` et `y` d'un type pouvant se ramener au type `point`, on retrouve le cas précédent, dans la mesure où la fonction membre `operator ==`, héritée de `point`, peut toujours s'appliquer à un objet de type `point` (c'est le cas des instructions 1, 2 et 4).

En revanche, si `x` est de type `int`, il n'est plus possible de lui appliquer une fonction membre. C'est ce qui se passe dans la dernière instruction, qui sera donc rejetée à la compilation.

N.B. Si l'unique argument de `operator=` était transmis par référence, la quatrième affectation serait rejetée, à moins d'avoir prévu en plus l'attribut `const`.

Exercice 108

Énoncé

Soit une classe `vect` permettant de manipuler des « vecteurs dynamiques » d'entiers (c'est-à-dire dont la dimension peut être fixée au moment de l'exécution) dont la déclaration (fournie dans le fichier `vect.h`) se présente ainsi :

```
class vect
{  int nelem ;               // nombre d'éléments
   int * adr ;               // adresse zone dynamique
                             // contenant les éléments
 public :
   vect (int) ;              // constructeur
                                    // (précise la taille du
vecteur)
   ~vect () ;                // destructeur
    int & operator [] (int) ; // accès à un élément par son
"indice"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire pour le nombre d'entiers reçu en argument et que l'opérateur `[]` peut être utilisé indifféremment dans une expression ou à gauche d'une affectation.

Créer une classe `vectb`, dérivée de `vect`, permettant de manipuler des vecteurs dynamiques, dans lesquels on peut fixer les « bornes » des indices, lesquelles seront fournies au constructeur de `vectb`. La classe `vect` apparaîtra ainsi comme un cas particulier de `vectb` (un objet de type `vect` étant un objet de type `vectb` dans lequel la limite inférieure de l'indice est `0`).

On ne se préoccupera pas, ici, des problèmes éventuellement posés par la recopie ou l'affectation d'objets de type `vectb`.

Solution

Nous prévoirons, dans `vectb`, deux membres donnée supplémentaires (`debut` et `fin`) pour conserver les bornes de l'indice (en toute rigueur, on pourrait se contenter d'un membre supplémentaire contenant la limite inférieure, sachant que la valeur supérieure pourrait s'en déduire à partir de la connaissance de la taille du vecteur ; toutefois, cette dernière information n'étant pas publique, nous rencontrerions des problèmes d'accès !).

Manifestement, `vectb` nécessite un constructeur à deux arguments entiers correspondant aux bornes de l'indice ; son en-tête pourrait commencer ainsi :

```
vectb (int d, int f)
```

Comme l'appel de ce constructeur entraînera automatiquement celui du constructeur de `vect`, il n'est pas question de faire l'allocation dynamique de notre vecteur dans `vectb`. Au contraire, nous réutilisons le travail effectué par `vect`, auquel nous transmettrons simplement le nombre d'éléments souhaités, c'est-à-dire ici `f - d + 1`. Voici l'en-tête complet du constructeur de `vectb` :

```
vectb (int d, int f) : vect (f-d+1)
```

La tâche spécifique de `vectb` se limitera à renseigner les valeurs des membres donnée `debut` et `fin`.

Aucun destructeur n'est nécessaire pour `vectb`, dans la mesure où son constructeur n'alloue aucun autre emplacement dynamique que celui alloué par `vect`.

En ce qui concerne l'opérateur `[]`, on peut penser que `vectb` l'hérite de `vect` et que, par conséquent, il n'est pas nécessaire de le surdéfinir. Toutefois, la

notation `t[i]` ne désigne plus forcément l'élément de rang `i` d'un objet de type `vectb`. Or, manifestement, on souhaitera qu'il en aille toujours ainsi. Il faut donc redéfinir `[]` pour `vectb`, quitte d'ailleurs à réutiliser l'opérateur défini dans `vect`.

Voici ce que pourrait être notre classe `vectb` (ici, on ne trouve qu'une définition, puisque les deux fonctions membre ont été définies en ligne) :

```
#include "vect.h"
class vectb : public vect
{   int debut, fin ;
  public :
    vectb ( int d, int f) : vect (f-d+1)
      { debut = d ; fin = f ;
      }
    int & operator [] (int i)
      { return vect::operator [] (i-debut) ;
      }
} ;
```

Remarques

1. Si le membre donnée `adr` avait été déclaré protégé (`protected`) dans la classe `vect`, nous aurions pu redéfinir l'opérateur `[]` de `vectb`, sans faire appel à celui de `vect` :

```
int & operator [] (int i)
  { return adr[i-debut] ;
  }
```

2. Aucune protection d'indices n'est à prévoir dans `vectb`, dès lors qu'elle a déjà été prévue dans `vect`.

Exercice 109

Énoncé

Soit une classe `int2d` (telle que celle créée dans l'exercice 93) permettant de manipuler des « tableaux dynamiques » d'entiers à deux dimensions dont la déclaration (fournie dans le fichier `int2d.h`) se présente ainsi :

```
class int2d
{  int nlig ;     // nombre de "lignes"
   int ncol ;     // nombre de "colonnes"
   int * adv ;    // adresse emplacement dynamique contenant
les valeurs
  public :
   int2d (int nl, int nc) ;       // constructeur
   ~int2d () ;                    // destructeur
   int & operator () (int, int) ; // accès à un élément, par
ses 2 "indices"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire et que l'opérateur `[]` peut être utilisé indifféremment dans une expression ou à gauche d'une affectation.

Créer une classe `int2db`, dérivée de `int2d`, permettant de manipuler des tableaux dynamiques, dans lesquels on peut fixer les « bornes » (valeur minimale et valeur maximale) des deux indices ; les quatre valeurs correspondantes seront fournies en arguments du constructeur de `int2db`.

On ne se préoccupera pas, ici, des problèmes éventuellement posés par la recopie ou l'affectation d'objets de type `int2db`.

Solution

Il suffit, en fait, de généraliser à la classe `int2d` le travail réalisé dans l'exercice précédent pour la classe `vect`. Voici ce que pourrait être notre classe `int2db` (ici, on ne trouve qu'une définition, dans la mesure où les fonctions membre de `int2db` ont été fournies en ligne) :

```
#include "int2d.h"
class int2db : public int2d
{   int ligdeb, ligfin ;       // bornes (mini, maxi) premier
indice
     int coldeb, colfin ;       // bornes (mini, maxi) second
indice
```

```
  public :                    // constructeur
    int2db (int ld, int lf, int cd, int cf) : int2d (lf-ld+1,
cf-cd+1)
      {  ligdeb = ld ; ligfin = lf ;
         coldeb = cd ; colfin = cf ;
      }
    int & int2db::operator () (int i, int j   // redéfinition
de operator ()
      {  return int2d::operator () (i-ligdeb, j-coldeb) ;
      }
} ;
```

Notez que, là non plus, aucune protection d'indice supplémentaire n'est à prévoir dans `int2db`, dès lors qu'elle a déjà été prévue dans `int2d`.

Exercice 110

Énoncé

Soit une classe `vect` permettant de manipuler des « vecteurs dynamiques » d'entiers, dont la déclaration (fournie dans un fichier `vect.h`) se présente ainsi (notez la présence de membres protégés) :

```
class vect
{ protected :      // en prévision d'une éventuelle classe
dérivée
    int nelem ;        // nombre d'éléments
     int * adr ;         // adresse zone dynamique contenant
les éléments
  public :
    vect (int) ;              // constructeur
    ~vect () ;                // destructeur
    int & operator [] (int) ;  // accès à un élément par son
"indice"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire et que l'opérateur `[]` peut être utilisé indifféremment dans une expression ou à gauche d'une affectation. En revanche, comme on peut le voir, cette classe n'a pas prévu de constructeur par recopie et elle n'a pas surdéfini l'opérateur d'affectation. L'affectation et la transmission par

valeur d'objets de type `vect` posent donc les « problèmes habituels ».

Créer une classe `vectok`, dérivée de `vect`, telle que l'affectation et la transmission par valeur d'objets de type `vectok` s'y déroulent convenablement. Pour faciliter l'utilisation de cette nouvelle classe, introduire une fonction membre `taille` fournissant la dimension d'un vecteur.

Écrire un petit programme d'essai.

Solution

Manifestement, la classe `vectok` n'a pas besoin de définir de nouveaux membres donnée. Pour déclarer des objets de type `vectok`, il faudra pouvoir en préciser la dimension, ce qui signifie que `vectok` devra absolument disposer d'un constructeur approprié. Ce dernier se contentera toutefois de retransmettre la valeur reçue en argument au constructeur de `vect` ; il aura donc un corps vide. Notez qu'il n'est pas nécessaire de prévoir un destructeur (car celui de `vect` sera appelé en cas de destruction d'un objet de type `vectok` et il n'y a rien de plus à faire).

Notez que pour gérer convenablement la recopie ou l'affectation d'objets, nous nous contenterons de la méthode déjà rencontrée qui consiste à dupliquer les objets concernés (en en faisant une « copie profonde »).

Pour satisfaire aux contraintes de l'énoncé, il nous faut donc prévoir de définir, dans `vectok`, un constructeur par recopie. Il faut alors tenir compte de ce que la recopie d'un objet de type `vectok` (qui fera donc appel à ce constructeur) entraînera alors l'appel du constructeur de `vect` qui sera indiqué dans l'en-tête du constructeur par recopie de `vectok`, du moins si une telle information est précisée (dans le cas contraire, il y aurait appel d'un constructeur sans argument de `vect`, ce qui n'est pas possible ici). Ici, nous disposons donc de deux possibilités :

- demander l'appel du constructeur par recopie de `vect`, ce qui conduit à cet en-tête :

  ```
  vectok::vectok (vectok & v) : vect (v)
  ```

  (rappelons que l'argument `v`, de type `vectok`, sera implicitement converti en type `vect`, pour pouvoir être transmis au constructeur `vect`).

Cette façon de faire conduit à la création d'un objet de type `vect`, obtenu par recopie (par défaut) de `v`. Il faudra alors compléter le travail en créant un nouvel emplacement dynamique pour le vecteur et en adaptant correctement la valeur de `adr`.

- demander l'appel du constructeur à un argument de `vect`, ce qui conduit à cet en-tête :

```
vectok::vectok (vectok & v) : vect (v.nelem)
```

Cette fois, il y aura création d'un nouvel objet de type `vect`, avec son propre emplacement dynamique, dans lequel il faudra néanmoins recopier les valeurs de `v`. C'est cette dernière solution que nous choisirons ici.

Toujours pour satisfaire aux contraintes de l'énoncé, nous devons surdéfinir l'affectation dans la classe `vectok`. Ici, aucun choix ne se présente. Nous utiliserons l'algorithme présenté dans l'exercice 89.

Voici ce que pourraient être la déclaration et la définition de notre classe `vectok` :

```
          /**** déclaration de la classe vectok ****/
#include "vect.h"
class vectok : public vect
{        // pas de nouveaux membres donnée
  public :
       vectok (int dim) : vect (dim)     // constructeur de
vectok : se contente
      {}                                 // de passer dim au
constructeur de vect
     vectok (vectok &) ;                 // constructeur par
recopie de vectok
       vectok & operator = (vectok &); // surdéfinition de
l'affectation de vectok
    int taille ()
       { return nelem ;
       }
} ;
    /***** définition du constructeur par recopie de vectok
*****/
    // il doit obligatoirement prévoir des arguments pour un
constructeur
     //      (quelconque) de vect (ici le constructeur à un
argument)
vectok::vectok (vectok & v) : vect (v.nelem) // ou const
vectok & v
```

```
{  int i ;
   for (i=0 ; i<nelem ; i++)  adr[i] = v.adr[i] ;
}
/***** définition de l'affectation entre vectok *****/
vectok & vectok::operator = (vectok & v)        // ou const
vectok & v
{  if (this != &v)
     { delete [] adr ;
       adr = new int [nelem = v.nelem] ;
       int i ;
       for (i=0 ; i<nelem ; i++) adr[i] = v.adr[i] ;
     }
   return (*this) ;
}
```

Voici un exemple de programme utilisant la classe `vectok`, accompagné du résultat de son exécution :

```
#include <iostream>   // voir N.B. du paragraphe Nouvelles
possibilités
                      // d'entrées-sorties du chapitre 2
using namespace std ;
#include "vectok.h"
main()
{  void fct (vectok) ;
   vectok v(6) ;
   int i ;
   for (i=0 ; i<v.taille() ; i++) v[i] = i ;
   cout << "vecteur v : " ;
   for (i=0 ; i<v.taille() ; i++) cout << v[i] << " " ;
   cout << "\n" ;
   vectok w(3) ;
   w = v ;
   cout << "vecteur w : " ;
   for (i=0 ; i<w.taille() ; i++) cout << w[i] << " " ;
   cout << "\n" ;
   fct (v) ;

}
void fct (vectok v)
{  cout << "vecteur reçu par fct : " << "\n" ;
   int i ;
   for (i=0 ; i<v.taille() ; i++) cout << v[i] << " " ;
}
```

```
 vecteur v : 0 1 2 3 4 5
 vecteur w : 0 1 2 3 4 5
 vecteur reçu par fct :
```

```
0 1 2 3 4 5
```

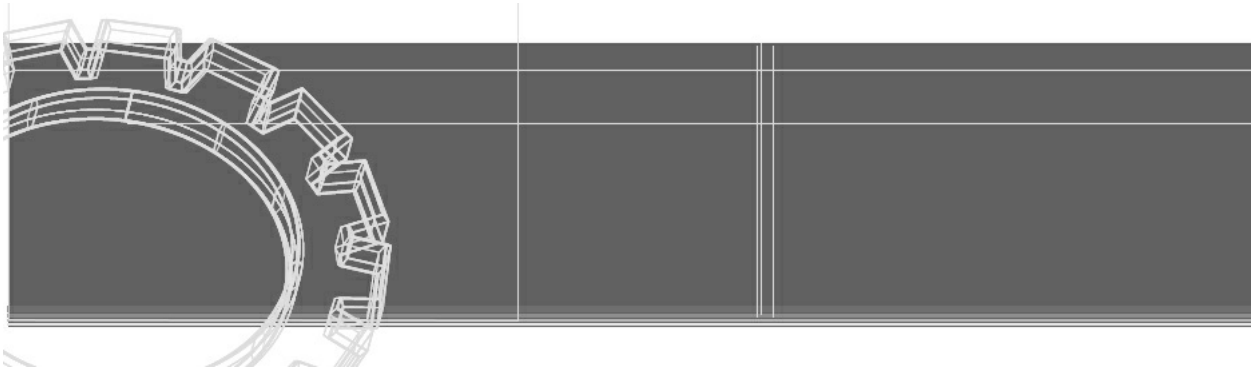
Remarque

Voici, à titre indicatif, ce que serait la définition du constructeur par recopie de `vectok`, dans le cas où l'on ferait appel au constructeur par recopie (par défaut) de `vect` :

```
    vectok::vectok (vectok & v) : vect (v)    // ou const
vectok & v
  {  nelem = v.nelem ;
     adr = new int [nelem] ;
     int i ;
     for (i=0 ; i<nelem ; i++)
        adr[i] = v.adr[i] ;
  }
```

Ici, il a fallu allouer un nouvel emplacement pour un vecteur, ce qui n'était pas le cas lorsque l'on faisait appel au constructeur à un argument de `vect` (puisque ce dernier faisait déjà une telle allocation).

Chapitre 14
L'héritage multiple



Rappels

Depuis la version 2.0, C++ autorise l'héritage multiple : une classe peut hériter de plusieurs autres classes, comme dans cet exemple où la classe `pointcol` hérite simultanément des classes `point` et `coul` :

```
class pointcol : public point, public coul // chaque dérivation, ici publique
                                           // pourrait être privée ou protégée
    {       // définition des membres supplémentaires (données ou fonctions)
            // ou redéfinition de membres existants déjà dans point ou coul
   } ;
```

Chacune des dérivations peut être publique, privée ou protégée. Les modalités d'accès aux membres de chacune des classes de base restent les mêmes que dans le cas d'une dérivation « simple ». L'opérateur de résolution de portée (`::`) peut être utilisé :

- soit lorsque l'on veut accéder à un membre d'une des classes de base, alors qu'il est redéfini dans la classe dérivée ;
- soit lorsque deux classes de base possèdent un membre de même nom et qu'il faut alors préciser celui qui nous intéresse.

Appel des constructeurs et des destructeurs

La création d'un objet entraîne l'appel du constructeur de chacune des classes de base, dans l'ordre où ces constructeurs sont mentionnés dans la déclaration de la classe dérivée (ici, `point` puis `coul` puisque nous avons écrit `class pointcol :` *`public point, public coul`*). Les destructeurs sont appelés dans l'ordre inverse.

Le constructeur de la classe dérivée peut mentionner, dans son en-tête, des informations à retransmettre à chacun des constructeurs des classes de base (ce sera généralement indispensable, sauf si une classe de base possède un constructeur sans argument ou si elle ne dispose pas du tout de constructeur). En voici un exemple :

```
pointcoul ( .....) : point (.....), coul (.....)
              |               |             |
              |               |             |
          arguments       arguments     arguments
          pointcoul         point          coul
```

Classes virtuelles

Par le biais de dérivations successives, il est tout à fait possible qu'une classe hérite deux fois d'une même classe. En voici un exemple dans lequel `D` hérite deux fois de `A` :

```
class B : public A
   { ..... } ;
class C : public A
   { ..... } ;
class D : public B, public C
   { ..... } ;
```

Dans ce cas, les membres donnée de la classe en question (`A` dans notre exemple) apparaissent **deux fois** dans la classe dérivée de deuxième niveau (ici `D`). Naturellement, il est nécessaire de faire appel à l'opérateur de

résolution de portée (`::`) pour lever l'ambiguïté. Si l'on souhaite que de tels membres n'apparaissent qu'une seule fois dans la classe dérivée de deuxième niveau, il faut, dans les déclarations des dérivées de premier niveau (ici `B` et `C`) déclarer avec l'attribut `virtual` la classe dont on veut éviter la duplication (ici `A`).

Voici comment on procéderait dans l'exemple précédent (le mot `virtual` peut être indifféremment placé avant ou après le mot `public` ou le mot `private`) :

```
class B : public virtual A
   { ..... } ;
class C : public virtual A
   { ..... } ;
class D : public B, public C
   { ..... } ;
```

Lorsqu'on a ainsi déclaré une classe virtuelle, il est nécessaire que les constructeurs d'éventuelles classes dérivées puissent préciser les informations à transmettre au constructeur de cette classe virtuelle (dans le cas usuel où l'on autorise la duplication, ce problème ne se pose plus ; en effet, chaque constructeur transmet les informations aux classes ascendantes dont les constructeurs transmettent, à leur tour, les informations aux constructeurs de chacune des occurrences de la classe en question – ces informations pouvant éventuellement être différentes). Dans ce cas, on le précise dans l'en-tête du constructeur de la classe dérivée, en plus des arguments destinés aux constructeurs des classes du niveau immédiatement supérieur, comme dans cet exemple :

```
D ( .....) : B (.....), C ( ..... ), A ( ..... )
      |           |           |            |
      |           |           |            |
  arguments    arguments   arguments    arguments
    de D        pour B      pour C       pour A
```

De plus, dans ce cas, les constructeurs des classes `B` et `C` (qui ont déclaré que `A` était « virtuelle ») n'auront plus à spécifier d'informations pour un constructeur de `A`.

Enfin, le constructeur d'une classe virtuelle est toujours appelé avant les autres.

Exercice 111

Énoncé

Quels seront les résultats fournis par ce programme :

```
#include <iostream>
using namespace std ;
class A
{   int n ;
    float x ;
  public :
    A (int p = 2)
    { n = p ; x = 1 ;
      cout << "** construction objet A : " << n << " " << x
<< "\n" ;
    }
} ;
class B
{   int n ;
    float y ;
  public :
    B (float v = 0.0)
    { n = 1 ;  y = v ;
      cout << "** construction objet B : " << n << " " << y
<< "\n" ;
    }
} ;
class C : public B, public A
{   int n ;
    int p ;
  public :
    C (int n1=1, int n2=2, int n3=3, float v=0.0) : A (n1),
B(v)
    { n = n3 ; p = n1+n2 ;
     cout << "** construction objet C : " << n << " " << p
<<"\n" ;
    }
} ;

main()
{  C c1 ;
   C c2 (10, 11, 12, 5.0) ;
}
```

Solution

L'objet `c1` est créé par appels successifs des constructeurs de `B`, puis de `A` (ordre imposé par la déclaration de la classe `C`, et non par l'en-tête du constructeur de `C` !). Le jeu de la transmission des arguments et des arguments par défaut conduit au résultat suivant :

```
** construction objet B : 1 0
** construction objet A : 1 1
** construction objet C : 3 3
** construction objet B : 1 5
** construction objet A : 10 1
** construction objet C : 12 21
```

Exercice 112

Énoncé

Même question que précédemment, en remplaçant simplement l'en-tête du constructeur de `C` par :

```
C (int n1=1, int n2=2, int n3=3, float v=0.0) :  B(v)
```

Solution

Ici, comme le constructeur de `C` n'a prévu aucun argument pour un éventuel constructeur de `A`, il y aura appel d'un constructeur sans argument, c'est-à-dire, en fait, appel du constructeur de `A`, avec toutes les valeurs prévues par défaut. Voici le résultat obtenu :

```
** construction objet B : 1 0
** construction objet A : 2 1
** construction objet C : 3 3
** construction objet B : 1 5
** construction objet A : 2 1
** construction objet C : 12 21
```

Exercice 113

Énoncé

Même question que dans l'exercice 58, en supposant que l'en-tête du constructeur de C est la suivante :

```
C (int n1=1, int n2=2, int n3=3, float v=0.0)
```

Solution

Cette fois, la construction d'un objet de type C entraînera l'appel d'un constructeur sans argument, à la fois pour B et pour A. Voici les résultats obtenus :

```
** construction objet B : 1 0
** construction objet A : 2 1
** construction objet C : 3 3
** construction objet B : 1 0
** construction objet A : 2 1
** construction objet C : 12 21
```

Exercice 114

Énoncé

Quels seront les résultats fournis par ce programme :

```
#include <iostream>
using namespace std ;
class A
{   int na ;
  public :
    A (int nn=1)
      { na = nn ; cout << "$$construction objet A " << na <<
"\n" ;
      }
} ;
```

```
class B : public A
{   float xb ;
  public :
    B (float xx=0.0)
      { xb = xx ; cout << "$$construction objet B " << xb <<
"\n" ;
      }
} ;
class C : public A
{   int nc ;
  public :
    C (int nn= 2) : A (2*nn+1)
      { nc = nn ;
        cout << "$$construction objet C " << nc << "\n" ;
      }
} ;
class D : public B, public C
{   int nd ;
  public :
    D (int n1, int n2, float x) : C (n1), B (x)
     { nd = n2 ;
       cout << "$$construction objet D " << nd << "\n" ;
     }
} ;

main()
{  D d (10, 20, 5.0) ;
}
```

Solution

La construction d'un objet de type `D` entraînera l'appel des constructeurs de `B` et de `C`, lesquels, avant leur exécution, appelleront chacun un constructeur de `A` : dans le cas de `B`, il y aura appel d'un constructeur sans argument (puisque l'en-tête de `B` ne mentionne pas de liste d'arguments pour `A`) ; en revanche, dans le cas de `C`, il s'agira (plus classiquement) d'un constructeur à un argument, comme mentionné dans l'en-tête de `C`.

Notez bien qu'il y a création de deux objets de type `A`. Voici les résultats obtenus :

```
$$construction objet A 1
$$construction objet B 5
$$construction objet A 21
$$construction objet C 10
```

```
$$construction objet D 20
```

Exercice 115

Énoncé

Transformer le programme précédent, de manière qu'un objet de type `D` ne contienne qu'une seule fois les membres de `A` (qui se réduisent en fait à l'entier `na`). On s'arrangera pour que le constructeur de `A` soit appelé avec la valeur `2*nn+1`, dans laquelle `nn` désigne l'argument du constructeur de `C`.

Solution

Dans la déclaration des classes `B` et `C`, il faut indiquer que la classe `A` est « virtuelle », de façon qu'elle ne soit incluse qu'une fois dans d'éventuelles descendantes de ces classes. D'autre part, le constructeur de `D` doit prévoir, outre les arguments pour les constructeurs de `B` et de `C`, ceux destinés à un constructeur de `A`.

En résumé, la déclaration de `A` reste inchangée, celle de `B` est transformée en :

```
class B : public virtual A
 {  // le reste est inchangé
 }
```

Celle de `C` est transformée de façon analogue :

```
class C : public virtual A
 {  // le reste est inchangé

 }
```

Enfin, dans `D`, l'en-tête du constructeur devient :

```
      D (int n1, int n2, float x) : C (n1), B (x), A (2*n1+1)
```

À titre indicatif, voici les résultats que fournirait le programme précédent ainsi transformé :

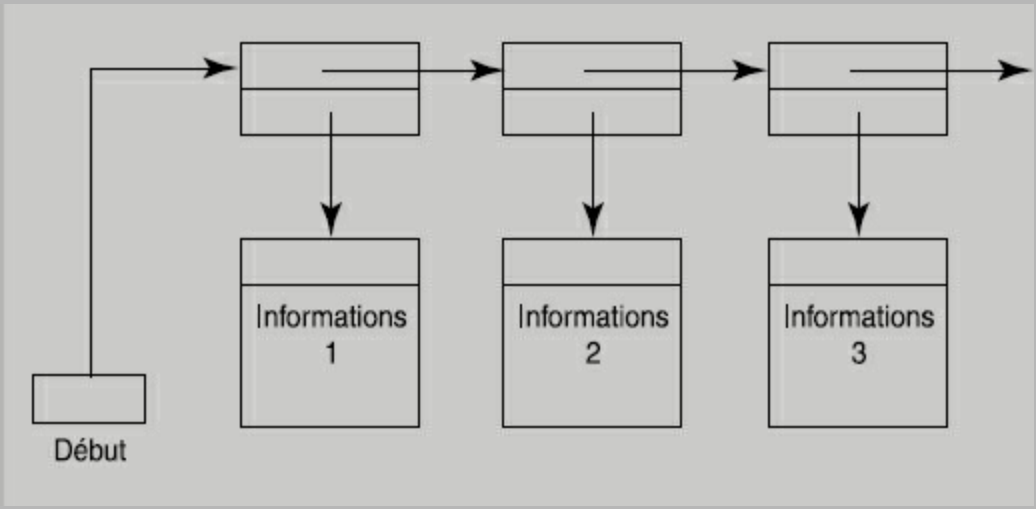
```
$$construction objet A 21
$$construction objet B 5
$$construction objet C 10
```

```
$$construction objet D 20
```

Exercice 116

Énoncé

On souhaite créer une classe `liste` permettant de manipuler des « listes chaînées » dans lesquelles la nature de l'information associée à chaque « nœud » de la liste n'est pas connue (par la classe). Une telle liste correspondra au schéma suivant :



La déclaration de la classe `liste` se présentera ainsi :

```
struct element                        // structure d'un élément
de liste
{ element * suivant ;                 // pointeur sur l'élément
suivant
  void * contenu ;                     // pointeur sur un objet
quelconque
} ;

class liste
{ element * debut ;                     // pointeur sur premier
élément
        // autres membres données éventuels
 public :
  liste () ;                         // constructeur
```

```
    ~liste () ;                        // destructeur
    void ajoute (void *) ;               // ajoute un élément en
début de
liste
    void * premier () ;                 // positionne sur premier
élément
    void * prochain () ;               // positionne sur prochain
élément
    int fini () ;
} ;
```

La fonction `ajoute` devra ajouter, en début de liste, un élément pointant sur l'information dont l'adresse est fournie en argument (`void *`). Pour « explorer » la liste, on a prévu trois fonctions :

- `premier`, qui fournira l'adresse de l'information associée au premier nœud de la liste et qui, en même temps, préparera le processus de parcours de la liste ;
- `prochain`, qui fournira l'adresse de l'information associée au « prochain nœud » ; des appels successifs de `prochain` devront permettre de parcourir la liste (sans qu'il soit nécessaire d'appeler une autre fonction) ;
- `fini`, qui permettra de savoir si la fin de liste est atteinte ou non.

1. Compléter la déclaration précédente de la classe `liste` et en fournir la définition de manière qu'elle fonctionne comme demandé.

2. Soit la classe `point` suivante :

```
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  void affiche () { cout << "Coordonnées : " << x << " " <<
y << "\n" ; }
} ;
```

Créer une classe `liste_points`, dérivée à la fois de `liste` et de `point`, pour qu'elle puisse permettre de manipuler des listes chaînées de points, c'est-à-dire des listes comparables à celles présentées ci-dessus, et dans lesquelles l'information associée est de type `point`. On devra pouvoir, notamment :

- ajouter un `point` en début d'une telle liste ;

- disposer d'une fonction membre `affiche` affichant les informations associées à chacun des points de la liste de points.

3. Écrire un petit programme d'essai.

Solutions

1. Manifestement, les fonctions `premier` et `prochain` nécessitent un « pointeur sur un élément courant ». Il sera membre donnée de la classe `liste`. Nous conviendrons (classiquement) que la fin de liste est matérialisée par un nœud comportant un pointeur « nul ». La classe `liste` devra disposer d'un constructeur dont le rôle se limitera à l'initialiser à une « liste vide », ce qui s'obtiendra simplement en plaçant un pointeur nul comme adresse de début de liste (cette façon de procéder simplifie grandement l'algorithme d'ajout d'un élément en début de liste, puisqu'elle évite d'avoir à distinguer des autres le cas de la liste vide).

   Comme un objet de type `liste` est amené à créer différents emplacements dynamiques, il est nécessaire de prévoir la libération de ces emplacements lorsque l'objet est détruit. Il faudra donc prévoir un destructeur, chargé de détruire les différents nœuds de la liste. À ce propos, notez qu'il n'est pas possible ici de demander au destructeur de détruire également les informations associées ; en effet, ce n'est pas l'objet de type `liste` qui a alloué ces emplacements : ils sont sous la responsabilité de l'utilisateur de la classe `liste`.

Voici ce que pourrait être notre classe `liste` complète :

```
#include <cstdlib>                  // pour NULL
struct element                      // structure d'un élément de
liste
{ element * suivant ;                  // pointeur sur l'élément
suivant
  void * contenu ;                   // pointeur sur un objet
quelconque
} ;
class liste
{ element * debut ;                   // pointeur sur premier
élément
  element * courant ;                 // pointeur sur élément
courant
 public :
```

```
  liste ()                          // constructeur
    { debut = NULL ;
      courant = debut ;             // par sécurité
    }
  ~liste () ;                       // destructeur
   void ajoute (void *) ;            // ajoute un élément en
début de liste
  void * premier ()                  // positionne sur premier
élément
  { courant = debut ;
    if (courant != NULL) return (courant->contenu) ;
                  else return NULL ;
  }
  void * prochain ()                 // positionne sur prochain
élément
  { if (courant != NULL)
  { courant = courant->suivant ;
       if (courant != NULL) return (courant->contenu) ;
  }
    return NULL ;
  }
  int fini () { return (courant == NULL) ; }
} ;
liste::~liste ()
{ element * suiv ;
  courant = debut ;
  while (courant != NULL )
    { suiv = courant->suivant ; delete courant ; courant =
suiv ; }
}
void liste::ajoute (void * chose)
{ element * adel = new element ;
  adel->suivant = debut ;
  adel->contenu = chose ;
  debut = adel ;
}
```

2. Comme nous le demande l'énoncé, nous allons donc créer une classe `liste_points` par :

```
class liste_points : public liste, public point
```

Notez que cet héritage, apparemment naturel, conduit néanmoins à introduire, dans la classe `liste_points`, deux membres donnée (`x` et `y`) n'ayant aucun intérêt par la suite.

En revanche, la création des fonctions membre demandées devient

extrêmement simple. En effet, la fonction d'insertion d'un point en début de liste peut être la fonction `ajoute` de la classe `liste` : nous n'aurons donc même pas besoin de la surdéfinir. En ce qui concerne la fonction d'affichage de tous les points de la liste (que nous nommerons également `affiche`), il lui suffira de faire appel :

– aux fonctions `premier`, `prochain` et `fini` de la classe `liste` pour le parcours de la liste de points ;

– à la fonction `affiche` de la classe `point` pour l'affichage d'un point.

Nous aboutissons à ceci :

```
class liste_points : public liste, public point
{ public :
   liste_points () {}
   void affiche () ;
} ;
void liste_points::affiche ()
{ point * ptr = (point *) premier() ;
  while ( ! fini() ) {  ptr->affiche () ; ptr = (point *)
prochain() ; }
}
```

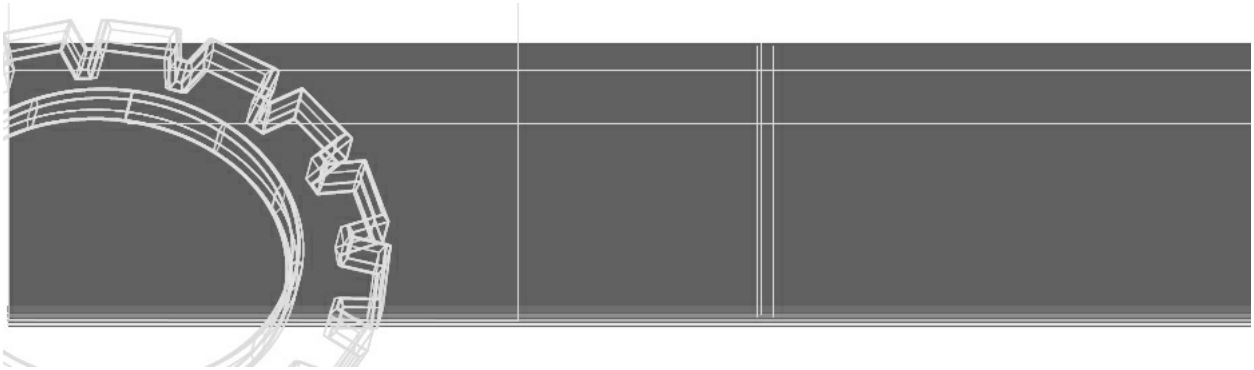
3. Exemple de programme d'essai :

```
#include "listepts.h"
main()
{ liste_points l ;
  point a(2,3), b(5,9), c(0,8) ;
  l.ajoute (&a) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&b) ; l.affiche () ; cout << "---------\n" ;
  l.ajoute (&c) ; l.affiche () ; cout << "---------\n" ;
}
```

À titre indicatif, voici les résultats fournis par ce programme :

```
Coordonnées : 2 3
---------
Coordonnées : 5 9
Coordonnées : 2 3
---------
Coordonnées : 0 8
Coordonnées : 5 9
Coordonnées : 2 3
---------
```

Chapitre 15
Les fonctions virtuelles



Rappels

Typage statique des objets (ou ligature dynamique des fonctions)

Les règles de compatibilité entre une classe de base et une classe dérivée permettent d'affecter à un pointeur sur une classe de base la valeur d'un pointeur sur une classe dérivée. Toutefois, par défaut, le type des objets pointés est défini lors de la compilation. Par exemple, avec :

```
class A
{  .....
   public :
     void fct (...) ;
     .....
} ;

A * pta ;
B * ptb ;
```

```
class B : public A
{  .....
   public :
     void fct (...) ;
     .....
} ;
```

une affectation telle que `pta = ptb` est autorisée. Néanmoins, quel que soit le contenu de `pta` (autrement dit, quel que soit l'objet pointé par `pta`), `pta->fct(...)` **appelle toujours la fonction** `fct` **de la classe** `A`.

Les fonctions virtuelles

L'emploi des fonctions virtuelles permet d'éviter les problèmes inhérents au typage statique. Lorsqu'une fonction est déclarée virtuelle (mot-clé `virtual`) dans une classe, les appels à une telle fonction ou à n'importe laquelle de ses redéfinitions dans des classes dérivées sont « résolus » au moment de l'exécution, selon le type de l'objet concerné. On parle de typage dynamique des objets (ou de ligature dynamique des fonctions). Par exemple, avec :

```
class A                          class B : public A
{  .....                         {  .....
   public :                         public :
     virtual void fct (...) ;         void fct (...) ;
     .....                            .....
};                                 } ;

A * pta ;
```

l'instruction `pta->fct (...)` appellera la fonction `fct` de la classe correspondant réellement au type de l'objet pointé par `pta`.

N.B. Il peut y avoir ligature dynamique même en dehors de l'utilisation de pointeurs (voyez, par exemple, l'exercice 118).

Règles

- Le mot-clé `virtual` ne s'emploie qu'une fois pour une fonction donnée ; plus précisément, il ne doit pas accompagner les redéfinitions de cette fonction dans les classes dérivées.
- Une méthode déclarée virtuelle dans une classe de base peut ne pas être redéfinie dans ses classes dérivées.
- Une fonction virtuelle peut être surdéfinie (chaque fonction surdéfinie pouvant être ou ne pas être virtuelle).
- Un constructeur ne peut pas être virtuel, un destructeur peut l'être.
- Par sa nature même, le mécanisme de ligature dynamique est limité à une

hiérarchie de classes ; souvent, pour qu'il puisse s'appliquer à toute une bibliothèque de classes, on sera amené à faire hériter toutes les classes de la bibliothèque d'une même classe de base.

Les fonctions virtuelles pures

Une « fonction virtuelle pure » se déclare avec une initialisation à zéro, comme dans :

```
virtual void affiche () = 0 ;
```

Lorsqu'une classe comporte au moins une fonction virtuelle pure, elle est considérée comme « abstraite », c'est-à-dire qu'il n'est plus possible de créer des objets de son type.

Une fonction déclarée virtuelle pure dans une classe de base peut ne pas être déclarée dans une classe dérivée et, dans ce cas, elle est à nouveau implicitement fonction virtuelle pure de cette classe dérivée (avant la version 3.0, une fonction virtuelle pure devait obligatoirement être redéfinie dans une classe dérivée ou déclarée à nouveau virtuelle pure).

Exercice 117

Énoncé

Quels résultats produira ce programme :

```
#include <iostream>
using namespace std ;
class point
{ protected :       // pour que x et y soient accessibles à
pointcol
   int x, y ;
  public :
   point (int abs=0, int ord=0) { x=abs ; y=ord ; }
   virtual void affiche ()
     { cout << "Je suis un point \n" ;
       cout << "   mes coordonnées sont : " << x << " " << y
<< "\n" ;
     }
```

```
} ;
class pointcol : public point
{  short couleur ;
  public :
   pointcol (int abs=0, int ord=0, short cl=1) : point (abs,
ord)
     { couleur = cl ;
     }
   void affiche ()
     { cout << "Je suis un point coloré \n" ;
        cout << "   mes coordonnées sont : " << x << " " <<
y ;
        cout << "   et ma couleur est :    " << couleur <<
"\n" ;
     }
} ;
main()
{
   point p(3,5) ; point * adp = &p ;
   pointcol pc (8,6,2) ; pointcol * adpc = &pc ;
   adp->affiche () ; adpc->affiche () ;      // instruction
1
   cout << "-----------------\n" ;
   adp = adpc ;
   adp->affiche () ; adpc->affiche () ;      // instruction
2
}
```

Dans les instructions 1, `adp` (de type `point *`) pointe sur un objet de type `point`, tandis que `adpc` (de type `pointcol *`) pointe sur un objet de type `pointcol`. Il y a appel de la fonction `affiche`, respectivement de `point` et de `pointcol` ; l'existence du « typage dynamique » n'apparaît pas clairement puisque, même en son absence (c'est-à-dire si la fonction `affiche` n'avait pas été déclarée virtuelle), on aurait obtenu le même résultat.

En revanche, dans les instructions 2, `adp` (de type `point *`) pointe maintenant sur un objet de type `pointcol`. Grâce au typage dynamique, `adp->affiche()` appelle bien la fonction `affiche` du type `pointcol`.

Voici les résultats complets fournis par ce programme :

```
Je suis un point
```

```
   mes coordonnées sont : 3 5
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
-----------------
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
Je suis un point coloré
   mes coordonnées sont : 8 6   et ma couleur est :    2
```

Exercice 118

Énoncé

Quels résultats produira ce programme :

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
 public :
  point (int abs=0, int ord=0) { x=abs ; y=ord ; }
  virtual void identifie ()
    { cout << "Je suis un point \n" ;
    }
  void affiche ()
    { identifie () ;
      cout << "Mes coordonnées sont : " << x << " " << y <<
"\n" ;
    }
} ;
class pointcol : public point
{ short couleur ;
 public :
  pointcol (int abs=0, int ord=0, int cl=1 ) : point (abs,
ord)
    { couleur = cl ; }
  void identifie ()
     { cout << "Je suis un point coloré de couleur : " <<
couleur << "\n" ;
    }
} ;
main()
{ point p(3,4) ;
```

```
    pointcol pc(5,9,5) ;
    p.affiche () ;
    pc.affiche () ;
    cout << "---------------\n" ;
    point * adp = &p ;
    pointcol * adpc = &pc ;
    adp->affiche () ; adpc->affiche () ;
    cout << "---------------\n" ;
    adp = adpc ;
    adp->affiche () ; adpc->affiche () ;
  }
```

Dans la fonction `affiche` de `point`, l'appel de `identifie` fait l'objet d'une ligature dynamique (puisque cette dernière fonction a été déclarée virtuelle). Lorsqu'un objet de type `pointcol` appelle une fonction `affiche`, ce sera bien la fonction `affiche` de `point` qui sera appelée (puisque `affiche` n'est pas redéfinie dans `pointcol`). Mais cette dernière fera appel, dans ce cas, à la fonction `identifie` de `pointcol`. Bien entendu, lorsqu'un objet de type `point` appelle `affiche`, cette dernière fera toujours appel à la fonction `identifie` de `point` (le même résultat serait obtenu sans ligature dynamique).

Voici les résultats complets fournis par notre programme :

```
Je suis un point
Mes coordonnées sont : 3 4
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
---------------
Je suis un point
Mes coordonnées sont : 3 4
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
---------------
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
Je suis un point coloré de couleur : 5
Mes coordonnées sont : 5 9
```

Exercice 119

Énoncé

On souhaite créer une classe nommée `ens_heter` permettant de manipuler des ensembles dont le type des éléments est non seulement inconnu de la classe, mais également susceptible de varier d'un élément à un autre. Pour que la chose soit possible, on imposera simplement la contrainte suivante : tous les types concernés devront dériver d'un même type de base nommé `base`. Le type `base` sera supposé connu au moment où l'on définit la classe `ens_heter`.

La classe `base` disposera au moins d'une fonction virtuelle pure nommée `affiche` ; cette fonction devra être redéfinie dans les classes dérivées pour afficher les caractéristiques de l'objet concerné.

La classe `ens_heter` disposera des fonctions membre suivantes :

- `ajoute` pour ajouter un nouvel élément à l'ensemble (elle devra s'assurer qu'il n'existe pas déjà) ;
- `appartient` pour tester l'appartenance d'un élément à l'ensemble ;
- `cardinal` qui fournira le nombre d'éléments de l'ensemble.

De plus, la classe devra être munie d'un « itérateur », c'est-à-dire d'un mécanisme permettant de parcourir les différents éléments de l'ensemble. On prévoira 3 fonctions :

- `init` pour initialiser le mécanisme d'itération ;
- `suivant` qui fournira en retour le prochain élément (objet d'un type dérivé de `base`) ;
- `existe` pour préciser s'il existe encore un élément non examiné.

Enfin, une fonction nommée `liste` permettra d'afficher les caractéristiques de tous les éléments de l'ensemble (elle fera, bien sûr, appel aux fonctions `affiche` des différents objets concernés).

On réalisera ensuite un petit programme d'essai de la classe `ens_heter`, en créant un ensemble comportant des objets de type `point` (deux

coordonnées entières) et `complexe` (une partie réelle et une partie imaginaire, toutes deux de type `float`). Naturellement, `point` et `complexe` devront dériver de `base`.

On ne se préoccupera pas des éventuels problèmes posés par l'affectation ou la transmission par valeur d'objets du type `ens_heter`.

N.B. Pour ne pas trop compliquer le programme, on fondera l'égalité de deux éléments d'un ensemble sur l'égalité de leurs adresses et non pas de leurs valeurs. Autrement dit, deux éléments de même type et de même valeur pourront appartenir à un même ensemble, à condition qu'il s'agisse bien de deux objets différents.

Manifestement, la classe `ens_heter` ne contiendra pas les objets correspondant aux éléments de l'ensemble, mais seulement des pointeurs sur ces différents éléments. À moins de fixer le nombre maximal d'éléments a priori, il est nécessaire de conserver ces pointeurs dans un emplacement alloué dynamiquement par le constructeur ; ce dernier comportera donc un argument permettant de préciser le nombre maximal d'éléments requis pour l'ensemble.

Outre l'adresse (`adel`) de ce tableau de pointeurs, on trouvera en membres donnée :

- le nombre maximal d'éléments (`nmax`) ;
- le nombre courant d'éléments (`nelem`) ;
- un entier (`courant`) qui servira au mécanisme d'itération (il désignera une adresse du tableau de pointeurs).

En ce qui concerne les fonctions `ajoute` et `appartient`, elles devront manifestement recevoir en argument un objet d'un type dérivé de `base`. Puisqu'on ne fait aucune hypothèse a priori sur la nature de tels objets, il est préférable d'éviter une transmission d'argument par valeur. Par souci de simplicité, nous choisirons une transmission par référence.

Les mêmes remarques s'appliquent à la valeur de retour de la fonction `suivant`.

Voici, en définitive, la déclaration de notre classe `ens_heter` :

```
         /*         fichier ensheter.H         */
         /* déclaration de la classe ens_heter */
class base ;
class ens_heter
{  int nmax ;       // nombre maximal d'éléments
   int nelem ;      // nombre courant d'éléments
    base * * adel ;   // adresse zone de pointeurs sur les
objets éléments
     int courant ;       // numéro d'élément courant (pour
l'itération)
 public :
   ens_heter (int=20) ;       // constructeur
   ~ens_heter () ;            // destructeur

   void ajoute (base &) ;     // ajout d'un élément
   int appartient (base &) ;  // appartenance d'un élément
   int cardinal () ;          // cardinal de l'ensemble
   void init () ;             // initialisation itération
   base & suivant () ;        // passage élément suivant
   int existe () ;
    void liste () ;              // affiche les "valeurs" des
différents éléments
} ;
```

La classe `base` (déclaration et définition) découle directement de l'énoncé :

```
class base
{ public :
    virtual void affiche () = 0 ;
} ;
```

Voici la définition des fonctions membre de `ens_heter` (notez que leur compilation nécessite la déclaration (définition) précédente de la classe `base` (et non pas seulement une simple indication de la forme `class base`) :

```
   /* définition de la classe ens_heter */
#include "ensheter.h"
ens_heter::ens_heter (int dim)
{  nmax = dim ;
   adel = new base * [dim] ;
   nelem = 0 ;
   courant = 0 ;       // précaution
}
ens_heter::~ens_heter ()
{  delete [] adel ;
}
void ens_heter::ajoute (base & obj)
```

```
{  if ((nelem < nmax) && (!appartient (obj))) adel [nelem++] =
& obj ;
}
int ens_heter::appartient (base & obj)
{  int trouve = 0 ;
   init () ;
    while ( existe () && !trouve)  if ( &suivant() == & obj)
trouve=1 ;
   return trouve ;
}
int ens_heter::cardinal ()
{   return nelem ;
}

void ens_heter::init ()
{  courant = 0 ;
}
base & ens_heter::suivant ()
{  if (courant<nelem) return (* adel [courant++]) ;
      // en pratique, il faudrait renvoyer un objet "bidon" si
fin
      // ensemble atteinte
}

int ens_heter::existe ()
{   return (courant<nelem) ;
}

void ens_heter::liste ()
{   init () ;
    while ( existe () )
       suivant () . affiche () ;
}
```

Voici un petit programme qui définit deux classes `point` et `complexe`, dérivées de `base` et qui crée un petit ensemble hétérogène (sa compilation nécessite la déclaration de la classe `base`). À la suite figure le résultat de son exécution :

```
#include "ens_heter.h"
#include "base.h"
#include <iostream>
using namespace std ;
class point : public base
{   int x, y ;
  public :
```

```
    point (int abs=0, int ord=0)
      { x = abs ; y = ord ;
      }
    void affiche ()
       { cout << "Point de coordonnées : " << x << " " << y <<
"\n" ;
      }
} ;

class complexe : public base
{   float re, im ;
  public :
    complexe (float reel=0.0, float imag=0.0)
      { re = reel ; im = imag ;
      }
    void affiche ()
      {  cout << "Complexe - partie réelle : " << re
             << ", partie imaginaire : " << im << "\n" ;
      }
} ;
   /* utilisation de la classe ens_heter */
main()
{   point p (1,3) ;
    complexe z (0.5, 3) ;
    ens_heter e ;
    cout << "cardinal de e : " << e.cardinal() << "\n" ;
    cout << "contenu de e  \n" ;
    e.liste () ;
    e.ajoute (p) ;
    cout << "cardinal de e : " << e.cardinal() << "\n" ;
    cout << "contenu de e  \n" ;
    e.liste () ;
    e.ajoute (z) ;
    cout << "cardinal de e : " << e.cardinal() << "\n" ;
    cout << "contenu de e  \n" ;
    e.liste () ;
    e.init () ; int n=0 ;
    while (e.existe()) { e.suivant() ;
                         n++ ;
                       }
      cout << "avec l'itérateur, on trouve : " << n << "
éléments\n" ;
}
```

```
cardinal de e : 0
contenu de e
```

```
cardinal de e : 1
contenu de e
Point de coordonnées : 1 3
cardinal de e : 2
contenu de e
Point de coordonnées : 1 3
Complexe - partie réelle : 0.5, partie imaginaire : 3
avec l'itérateur, on trouve : 2 éléments
```

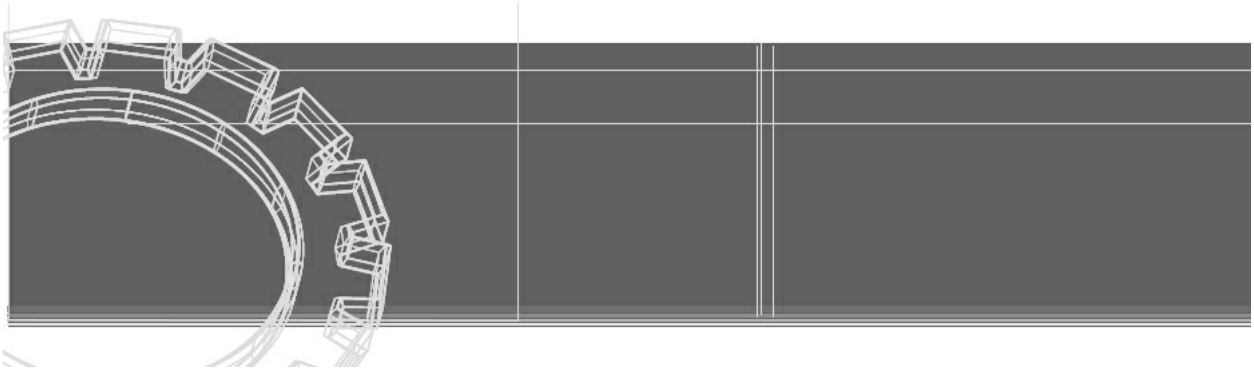
Remarque

Si l'on cherchait à résoudre les problèmes posés par l'affectation et la transmission par valeur d'objets de type `ens_heter`, on serait amené à effectuer des « copies complètes » (on dit souvent « profondes ») de l'objet, c'est-à-dire tenant compte non seulement de ses membres, mais aussi de ses parties dynamiques.

Cela conduirait effectivement à recopier la partie dynamique de l'ensemble, c'est-à-dire le tableau de pointeurs d'adresse `base` *. Mais que faudrait-il faire pour les éléments eux-mêmes (pointés par les différentes valeurs du tableau) ? Même si l'on admet qu'il faut également les dupliquer, il n'est pas possible de le faire sans connaître la « structure » des objets concernés.

D'une manière générale, vous voyez que cet exemple, par les questions qu'il soulève, plaide largement en faveur de la constitution de bibliothèques d'objets, dans lesquelles sont définies, a priori, un certain nombre de règles communes. Parmi ces règles, on pourrait notamment imposer à chaque objet de posséder une fonction (de nom unique) en assurant la copie profonde ; naturellement, pour pouvoir l'utiliser correctement, il faudrait que la ligature dynamique soit possible, c'est-à-dire que tous les objets concernés dérivent d'un même objet de base.

Chapitre 16
Les flots d'entrée et de sortie



Rappels

Un flot est un canal recevant (flot d'« entrée ») ou fournissant (flot de « sortie ») de l'information. Ce canal est associé à un périphérique ou à un fichier. Un flot d'entrée est un objet de type `istream` tandis qu'un flot de sortie est un objet de type `ostream`. Le flot `cout` est un flot de sortie prédéfini, connecté à la sortie standard `stdout` ; de même, le flot `cin` est un flot d'entrée prédéfini, connecté à l'entrée standard `stdin`.

La classe ostream

Elle surdéfinit l'opérateur << sous la forme d'une fonction membre :

```
ostream & operator << (expression)
```

L'expression correspondant à son deuxième opérande peut être d'un type de base quelconque, y compris `char`, `char *` (on obtient la chaîne pointée) ou un pointeur sur un type quelconque autre que `char` (on obtient la valeur du pointeur) ; pour obtenir la valeur de l'adresse d'une chaîne, on la convertit

artificiellement en un pointeur de type `void *`.

Fonctions membre :

- **`ostream & put (char c)`** transmet au flot correspondant le caractère `c`.
- **`ostream & write (void * adr, int long)`** envoie `long` caractères, prélevés à partir de l'adresse `adr`.

La classe istream

Elle surdéfinit l'opérateur >> sous la forme d'une fonction membre :

```
istream & operator >> (type_de_base &)
```

Le `type_de_base` peut être quelconque, pour peu qu'il ne s'agisse pas d'un pointeur (`char *` est cependant accepté ; il correspond à l'entrée d'une chaîne de caractères, et non d'une adresse).

Les « espaces blancs » (espace, tabulation horizontale `\t` ou verticale `\v`, fin de ligne `\n` et changement de page `\f`) servent de « délimiteurs » (comme dans `scanf`), y compris pour les chaînes de caractères.

Principales fonctions membre :

- **`istream & get (char & c)`** extrait un caractère du flot d'entrée et le range dans `c`.
- **`int get ()`** extrait un caractère du flot d'entrée et en renvoie la valeur (sous forme d'un entier) ; fournit EOF en cas de fin de fichier.
- **`istream & read (void * adr, int taille)`** lit `taille caractères` sur le flot et les range à partir de l'adresse `adr`.

La classe iostream

Elle est dérivée de `istream` et `ostream`. Elle permet de réaliser des entrées sorties « conversationnelles ».

Le statut d'erreur d'un flot

À chaque flot est associé un ensemble de bits d'un entier formant le « statut d'erreur du flot ».

Les bits d'erreur

La classe `ios` (dont dérivent `istream` et `ostream`) définit les constantes suivantes :

- `eofbit` : fin de fichier (le flot n'a plus de caractères disponibles) ;
- `failbit` : la prochaine opération sur le flot ne pourra pas aboutir ;
- `badbit` : le flot est dans un état irrécupérable ;
- `goodbit` (valant, en fait `0`) : aucune des errreurs précédentes.

Une opération sur le flot a réussi lorsqu'un des bits `goodbit` ou `eofbit` est activé. La prochaine opération sur le flot ne pourra réussir que si `goodbit` est activé (mais il n'est pas certain qu'elle réussisse !).

Lorsqu'un flot est dans un état d'erreur, aucune opération ne peut aboutir tant que la condition d'erreur n'a pas été corrigée et que le bit d'erreur correspondant n'a pas été remis à zéro (à l'aide de la fonction `clear`).

Accès aux bits d'erreur

La classe `ios` contient cinq fonctions membre :

- **`eof ()`** : valeur de `eofbit` ;
- **`bad ()`** : valeur de `badbit` ;
- **`fail ()`** : valeur de `failbit ;`
- **`good ()`** : `1` si aucun bit du statut d'erreur n'est activé ;
- **`rdstate ()`** : valeur du statut d'erreur (entier).

Modification du statut d'erreur

`void clear (int i=0)` donne la valeur `i` au statut d'erreur. Pour activer un seul bit (par exemple `badbit`), on procédera ainsi (`fl` étant un flot) :

```
fl.clear (ios::badbit | fl rdstate() ) ;
```

Surdéfinition de () et de !

Si `fl` est un flot, `(fl)` est vrai si aucun des bits d'erreur n'est activé (c'est-à-dire si `good` est vrai) ; de même, `!fl` est vrai si un des bits d'erreur précédents est activé (c'est-à-dire si `good` est faux).

Surdéfinition de << et >> pour des types classe

On surdéfinira << et >> pour une classe quelconque, sous forme de fonctions amies, en utilisant ces « canevas » :

```
ostream & operator << (ostream sortie, type_classe objet[1])
{
       // Envoi sur le flot sortie des membres de objet en
utilisant
      // les possibilités classiques de << pour les types de
base
     // c'est-à-dire des instructions de la forme :
     // sortie << ..... ;
   return sortie ;
}
istream & operator >> (istream & entree, type_de_base & objet)
{
      // Lecture des informations correspondant aux différents
membres de objet
      // en utilisant les possibilités classiques de >> pour
les types de base
     // c'est-à-dire des instructions de la forme :
     // entree >> ..... ;
   return entree ;
}
```

Le mot d'état du statut de formatage

À chaque flot est associé un « statut de formatage », constitué d'un mot d'état et de trois valeurs numériques (gabarit, précision et caractère de remplissage).

Voici les principaux bits du mot d'état :

Le mot d'état du statut de formatage (partiel)

Nom de champ	Nom du bit (s'il existe)	Signification (quand activé)
`ios::basefield`	`ios::dec`	conversion décimale
	`ios::oct`	conversion octale
	`ios::hex`	conversion hexadécimale
	`ios::showbase`	affichage indicateur de base (en sortie)

	`ios::showpoint`	affichage point décimal (en sortie)
`ios::floatfield`	`ios::scientific`	notation « scientifique »
	`ios::fixed`	notation « point fixe »

Action sur le statut de formatage

On peut utiliser, soit des « manipulateurs » qui peuvent être « simples » ou « paramétriques », soit des fonctions membre.

a. Les manipulateurs non paramétriques

Ils s'emploient sous la forme :

```
flot << manipulateur
flot >> manipulateur
```

Les principaux manipulateurs non paramétriques sont :

Les principaux manipulateurs non paramétriques

Manipulateur	Utilisation	ACTION
`dec`	`Entrée/Sortie`	Active le bit de conversion décimale
`hex`	`Entrée/Sortie`	Active le bit de conversion hexadécimale
`oct`	`Entrée/Sortie`	Active le bit de conversion octale
`endl`	`Sortie`	Insère un saut de ligne et vide le tampon
`ends`	`Sortie`	Insère un caractère de fin de chaîne `(\0)`

b. Les manipulateurs paramétriques

Ils s'utilisent sous la forme :

```
istream & manipulateur (argument)
ostream & manipulateur (argument)
```

Voici les principaux manipulateurs paramétriques.

Les principaux manipulateurs paramétriques

Manipulateur	Utilisation	Rôle
`setbase (int)`	`Entrée/Sortie`	Définit la base de conversion
`setprecision (int)`	`Entrée/Sortie`	Définit la précision des nombres flottants
`setw (int)`	`Entrée/Sortie`	Définit le gabarit. Il retombe à `0` après chaque opération

L'utilisation des manipulateurs paramétriques nécessite l'inclusion du fichier en-tête `iomanip`. Dans certaines implémentations, il peut encore s'agir de `iomanip.h` (associé alors à `iostream.h` et non à `iostream`, et sans l'utilisation de l'espace de noms `std`).

Association d'un flot à un fichier

La classe `ofstream`, dérivant de `ostream`, permet de créer un flot de sortie associé à un fichier :

```
ofstream flot (char * nomfich, mode_d_ouverture)
```

La fonction membre `seekp (déplacement, origine)` permet d'agir sur le pointeur de fichier.

De même, la classe `ifstream`, dérivant de `istream`, permet de créer un flot d'entrée associé à un fichier :

```
ifstream flot (char * nomfich, mode_d_ouverture)
```

La fonction membre `seekg (déplacement, origine)` permet d'agir sur le pointeur de fichier.

Dans tous les cas, la fonction `close` permet de fermer le fichier.

L'utilisation des classes `ofstream` et `ifstream` demande l'inclusion du fichier `fstream`. Dans certaines implémentations, il peut encore s'agir de `fstream.h` (associé alors à `iostream.h` et non à `iostream`, et sans l'utilisation de l'espace de noms `std`).

Modes d'ouverture d'un fichier

Les différents modes d'ouverture d'un fichier

Bit de mode d'ouverture	Action
`ios::in`	Ouverture en lecture (obligatoire pour la classe `ifstream`)
`ios::out`	Ouverture en écriture (obligatoire pour la classe `ofstream`)
`ios::app`	Ouverture en ajout de données (écriture en fin de fichier)
`ios::ate`	Se place en fin de fichier après ouverture
`ios::trunc`	Si le fichier existe, son contenu est perdu (obligatoire si `ios::out` est activé sans `ios::ate` ni `ios::app`)
`ios::binary`	(Utile dans certaines implémentations uniquement.) Le fichier est ouvert en mode dit « binaire » ou encore « non translaté »

Exercice 120

Énoncé

Écrire un programme qui lit un nombre réel et qui en affiche le carré sur un « gabarit » minimal de 12 caractères, de 22 façons différentes :

- en « point fixe », avec un nombre de décimales variant de 0 à 10,
- en notation scientifique, avec un nombre de décimales variant de 0 à 10.

Dans tous les cas, on affichera les résultats avec cette présentation :

```
précision de xx chiffres : ccccccccccc
```

Solution

Il faut donc activer d'abord le bit `fixed`, ensuite le bit `scientific` du champ `floatfield`. Nous utiliserons la fonction `setf`, membre de la classe `ios`. Notez bien qu'il faut éviter d'écrire, par exemple :

```
setf (ios::fixed) ;
```

En effet, cela activerait le bit `fixed`, sans modifier les autres donc, en particulier, sans modifier les autres bits du champ `floatfield`.

Le gabarit d'affichage est déterminé par le manipulateur `setw`. Notez qu'il faut transmettre ce manipulateur au flot concerné, juste avant d'afficher l'information voulue.

```
#include  <iomanip>            //  pour  les  "manipulateurs
paramétriques"
#include <iostream>     // voir N.B. du paragraphe Nouvelles
possibilités
                       // d'entrées-sorties du chapitre 2
using namespace std ;
main()
{  float val, carre ;
   cout << "donnez un nombre réel : " ;
   cin  >> val ;
   carre = val*val ;
   cout << "Voici son carré : \n" ;
   int i ;
   cout << "  en notation point fixe : \n" ;
    cout.setf (ios::fixed, ios::floatfield) ;// met à 1 le bit
ios::fixed
                                                    // du champ
ios::floatfield
   for (i=0 ; i<10 ; i++)
       cout << "       précision de " << setw (2) << i << "
chiffres : "
          << setprecision (i) << setw (12) << carre << "\n" ;
   cout << "  en notation scientifique : \n" ;
   cout.setf (ios::scientific, ios::floatfield) ;
   for (i=0 ; i<10 ; i++)
       cout << "       précision de " << setw (2) << i << "
chiffres : "
          << setprecision (i) << setw (12) << carre << "\n" ;
}
```

À titre indicatif, voici un exemple d'exécution de ce programme :

```
donnez un nombre réel : 12.3456
Voici son carré :
  en notation point fixe :
      précision de  0 chiffres :          152
      précision de  1 chiffres :        152.4
      précision de  2 chiffres :       152.41
      précision de  3 chiffres :      152.414
      précision de  4 chiffres :     152.4138
      précision de  5 chiffres :    152.41385
```

```
     précision de  6 chiffres :   152.413849
     précision de  7 chiffres :  152.4138489
     précision de  8 chiffres : 152.41384888
     précision de  9 chiffres : 152.413848877
en notation scientifique :
     précision de  0 chiffres : 1.524138e+002
     précision de  1 chiffres :     1.5e+002
     précision de  2 chiffres :    1.52e+002
     précision de  3 chiffres :   1.524e+002
     précision de  4 chiffres :  1.5241e+002
     précision de  5 chiffres : 1.52414e+002
     précision de  6 chiffres : 1.524138e+002
     précision de  7 chiffres : 1.5241385e+002
     précision de  8 chiffres : 1.52413849e+002
     précision de  9 chiffres : 1.524138489e+002
```

Exercice 121

Énoncé

Soit la classe `point` suivante :

```
class point
{   int x, y ;
   public :
    // fonctions membre
} ;
```

Surdéfinir les opérateurs << et >> de manière qu'il soit possible de lire un point sur un flot d'entrée ou d'écrire un point sur un flot de sortie. On prévoira qu'un tel point soit représenté sous la forme :

```
<entier, entier>
```

avec éventuellement des séparateurs « espaces_blancs » supplémentaires, de part et d'autre des nombres entiers.

Solution

Nous devons donc surdéfinir les opérateurs << et >> pour qu'ils puissent recevoir, en deuxième opérande, un argument de type `point`. Il ne pourra

s'agir que de fonctions amies, dont les prototypes se présenteront ainsi :

```
ostream & operator << (ostream &, point) ;
istream & operator >> (istream &, point) ;
```

L'écriture de `operator <<` ne présente pas de difficultés particulières : on se contente d'écrire, sur le flot concerné, les coordonnées du point, accompagnées des symboles < et >.

En revanche, l'écriture de `operator >>` nécessite un peu plus d'attention. En effet, il faut s'assurer que l'information se présente bien sous la forme requise et, si ce n'est pas le cas, prévoir de donner au flot concerné l'état `bad`, afin que l'utilisateur puisse savoir que l'opération s'est mal déroulée (en testant « naturellement » l'état du flot).

Voici ce que pourraient être la déclaration de notre classe (nous l'avons simplement munie d'un constructeur) et la définition des deux fonctions amies voulues :

```
#include <iostream>
using namespace std ;

class point
{
   int x, y ;
  public :
   point (int abs=0, int ord=0)
     { x = abs ; y = ord ; }
   int abscisse () { return x ; }
   friend ostream & operator << (ostream &, point) ;
   friend istream & operator >> (istream &, point &) ;
} ;

ostream & operator << (ostream & sortie, point p)
{
  sortie << "<" << p.x << "," << p.y << ">" ;
  return sortie ;
}

istream & operator >> (istream & entree, point & p)
{
   char c = '\0' ;
   float x, y ;
   int ok = 1 ;
   entree >> c ;
   if (c != '<') ok = 0 ;
```

```
      else
       { entree >> x >> c ;
         if (c != ',') ok = 0 ;
            else
               { entree >> y >> c ;
                 if (c != '>') ok = 0 ;
               }
       }
   if (ok) { p.x = x ; p.y = y ; } // on n'affecte à p que si
tout est OK
      else entree.clear (ios::badbit | entree.rdstate () ) ;
   return entree ;
}
```

À titre indicatif, voici un petit programme d'essai, accompagné d'un exemple d'exécution :

```
main()
{
  char ligne [121] ;
  point a(2,3), b ;
  cout << "point a : " << a << "  point b : " << b << "\n" ;
  do
   { cout << "donnez un point : " ;
     if (cin >> a)  cout << "merci pour le point : " << a <<
"\n" ;
             else { cout << "** information incorrecte \n" ;
                    cin.clear () ;
                    cin.getline (ligne, 120, '\n') ;
                  }
   }
  while ( a.abscisse () ) ;
}
```

```
point a : <2,3>  point b : <0,0>
donnez un point : 4,5
** information incorrecte
donnez un point : <4,5<
** information incorrecte
donnez un point : <4,5>
merci pour le point : <4,5>
donnez un point : < 8,   9     >
merci pour le point : <8,9>
donnez un point : bof
** information incorrecte
donnez un point : <0,0>
```

```
merci pour le point : <0,0>
```

Exercice 122

Énoncé

Écrire un programme qui enregistre (sous forme « binaire », et non pas formatée), dans un fichier de nom fourni par l'utilisateur, une suite de nombres entiers fournis sur l'entrée standard. On conviendra que l'utilisateur fournira la valeur `0` (qui ne sera pas enregistrée dans le fichier) pour préciser qu'il n'a plus d'entiers à entrer.

Solution

Si `nomfich` désigne une chaîne de caractères, la déclaration :

```
ofstream sortie (nomfich, ios::out) ;
```

permet de créer un flot de nom `sortie`, de l'associer au fichier dont le nom figure dans `nomfich` et d'ouvrir ce fichier en écriture.

L'écriture dans le fichier en question se fera par la fonction `write`, appliquée au flot `sortie`.

Voici le programme demandé :

```
const int LGMAX = 20 ;
#include <cstdlib>                          // pour exit
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std ;
main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à créer : " ;
   cin >> setw (LGMAX) >> nomfich ;
   ofstream sortie (nomfich, ios::out) ;
   if (!sortie) { cout << "création impossible \n" ;
```

```
                exit (1) ;
              }
   do
      { cout << "donnez un entier : " ;
        cin >> n ;
        if (n) sortie.write ((char *)&n, sizeof(int) ) ;
      }
   while (n && sortie) ;
   sortie.close () ;
}
```

Notez que `if (!sortie)` est équivalent à `if (!sortie.good())` et que `while (n && sortie)` est équivalent à `while (n && sortie.good())`.

Exercice 123

Énoncé

Écrire un programme permettant de lister (sur la sortie standard) les entiers contenus dans un fichier tel que celui créé par l'exercice précédent.

Solution

```
const int LGMAX = 20 ;
#include <cstdlib>                          // pour exit
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std ;
main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier à lister : " ;
   cin >> setw (LGMAX) >> nomfich ;
   ifstream entree (nomfich, ios::in) ;
   if (!entree) { cout << "ouverture impossible \n" ;
                  exit (1) ;
                }
   while ( entree.read ( (char*)&n, sizeof(int) ) )
```

```
        cout << n << "\n" ;
   entree.close () ;
}
```

Exercice 124

Énoncé

Écrire un programme permettant à un utilisateur de retrouver, dans un fichier tel que celui créé dans l'exercice 122, les entiers dont il fournit le « rang ». On conviendra qu'un rang égal à 0 signifie que l'utilisateur souhaite mettre fin au programme.

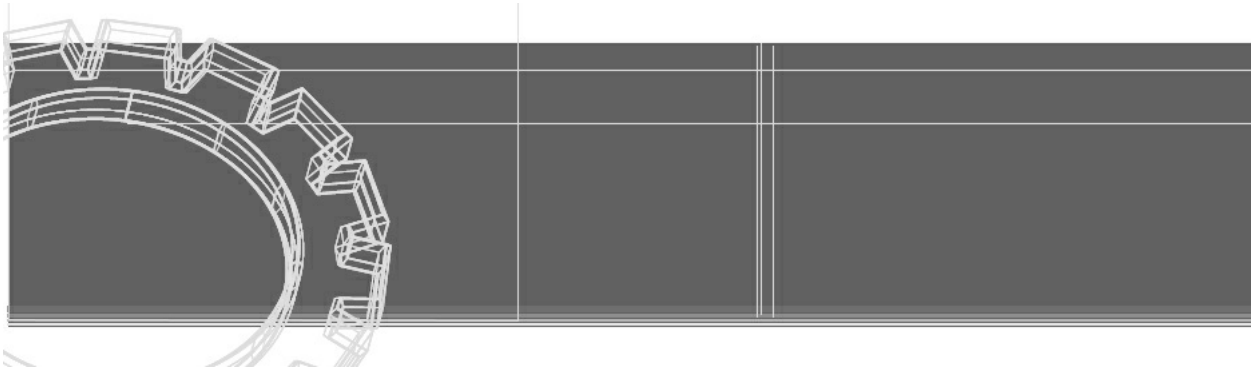
Solution

```
const int LGMAX_NOM_FICH = 20 ;
#include <cstdlib>               // pour exit
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std ;

main()
{
  char nomfich [LGMAX_NOM_FICH + 1] ;
  int n, num ;
  cout << "nom du fichier à consulter : " ;
  cin >> setw (LGMAX_NOM_FICH) >> nomfich ;
  ifstream entree (nomfich, ios::in) ;
  if (!entree) { cout << "Ouverture impossible\n" ;
                 exit (1) ;
               }
  do
    { cout << "Numéro de l'entier recherché : " ;
      cin >> num ;
      if (num)
         { entree.seekg (sizeof(int) * (num-1) , ios::beg ) ;
           entree.read ( (char *) &n, sizeof(int) ) ;
           if (entree) cout << "-- Valeur : " << n << "\n" ;
              else {  cout << "-- Erreur\n" ;
```

```
                    entree.clear () ;
                  }
            }
      }
    while (num) ;
    entree.close () ;
  }
```

1. Ici, la transmission peut se faire par valeur ou par référence.

Chapitre 17
Les patrons de fonctions



Rappels

Introduite par la version 3, la notion de patron de fonctions permet de définir ce qu'on nomme souvent des « fonctions génériques ». Plus précisément, à l'aide d'une unique définition comportant des « paramètres de type », on décrit toute une famille de fonctions ; le compilateur « fabrique » (on dit aussi « instancie ») la ou les fonctions nécessaires à la demande (on nomme souvent ces instances « fonctions patron »).

La version 3 limitait les paramètres d'un patron de fonctions à des paramètres de type. La norme ANSI a, en outre, introduit les « paramètres expression ».

Définition d'un patron de fonctions

On précise les paramètres (muets) de type, en faisant précéder chacun du mot (relativement arbitraire) `class` sous la forme `template <class ..., class ..., ...>`. La définition de la fonction est classique, hormis le fait que les paramètres muets de type peuvent être employés n'importe où un type

effectif est permis.

Par exemple :

```
template <class T, class U> void fct (T a, T * b, U c)
{
   T x ;               // variable locale x de type T
   U *  adr ;          // variable locale adr de type U *
   ...
   adr = new T [10] ;  // allocation tableau de 10 éléments de
type T
   ...
   n = sizeof (T) ;    // une instruction utilisant le type T
   ...
}
```

Une instruction telle que (`T` désignant un type quelconque) :

```
T x (3) ;
```

est légale même si `T` n'est pas un type classe ; dans ce dernier cas, elle est simplement équivalente à :

```
T x = 3 ;
```

Instanciation d'une fonction patron

Chaque fois qu'on utilise une fonction ayant un nom de patron, le compilateur cherche à utiliser ce patron pour créer (instancier) une fonction adéquate. Pour ce faire, il cherche à réaliser une **correspondance absolue** des types : aucune conversion, qu'il s'agisse de promotion numérique ou de conversion standard n'est permise ; qui plus est, les qualifieurs `const` et `volatile` doivent être exactement les mêmes.

Voici des exemples utilisant notre patron précédent :

```
int n, p ; float x ; char c ;
int * adi ; float * adf ;
class point ; point p ; point * adp ;

fct (n, adi, x) ;    // instancie la fonction void fct (int,
int *, float)
```

```
fct (n, adi, p)      // instancie la fonction void fct (int,
int *, int)
fct (x, adf, p) ;   // instancie la fonction void fct (float,
float *, int)
fct (c, adi, x) ;   // erreur char et int * ne correspondent
pas à T et T*
                   // (pas de conversion)
fct (&n, &adi, x) ; // instancie la fonction void fct (int *,
int * *, float)
fct (p, adp, n) ;   // instancie la fonction void fct (point,
point *, int)
```

D'une manière générale, il est nécessaire que **chaque paramètre de type** apparaisse **au moins une fois dans l'en-tête** du patron.

La définition d'un patron de fonctions ne peut pas être compilée seule ; de toute façon, elle doit être connue du compilateur pour qu'il puisse instancier la bonne fonction patron. En général, les définitions de patrons de fonctions figureront dans des fichiers d'extension `h`, de façon à éviter d'avoir à en fournir systématiquement la liste.

Les paramètres expression d'un patron de fonctions

Ils ont été introduits par la norme. Un paramètre expression d'un patron de fonctions se présente comme un argument usuel de fonction ; il n'apparaît pas dans la liste de paramètres de type (`template`) et il doit apparaître dans l'en-tête du patron. Par exemple :

```
template <class T> int compte (T * tab, int n)
{  // ici, on peut se servir de la valeur de l'entier n
     // comme on le ferait dans n'importe quelle fonction
ordinaire
}
```

Un patron de fonctions peut disposer d'un ou de plusieurs paramètres expression. Lors de l'appel, leur type n'a plus besoin de correspondre exactement à celui attendu : il suffit qu'il soit acceptable par affectation, comme dans n'importe quel appel d'une fonction ordinaire.

Surdéfinition de patrons de fonctions et spécialisation de fonctions de patrons

On peut définir plusieurs patrons de même nom, possédant des paramètres (de type ou expression) différents. La seule règle à respecter dans ce cas est que l'appel d'une fonction de ce nom ne doit pas conduire à une ambiguïté : un seul patron de fonctions doit pouvoir être utilisé à chaque fois.

Par ailleurs, il est possible de fournir la définition d'une ou plusieurs fonctions particulières qui seront utilisées en lieu et place de celle instanciée par un patron. Par exemple, avec :

```
template <class T> T min (T a, T b)   // patron de fonctions
{ ... }
char * min (char * cha, char * chb)   // version spécialisée
pour le type char *
{ ... }
int n, p;
char * adr1, * adr2 ;

min (n, p)          // appelle la fonction instanciée par le
patron général
                    // soit ici  :    int min (int, int)
min (adr1, adr2)    // appelle la fonction spécialisée
                    // char * min (char *, char *)
```

Algorithme d'instanciation ou d'appel d'une fonction

Précisons comment doivent être aménagées les règles de recherche d'une fonction surdéfinie, dans le cas où il existe un ou plusieurs patrons de fonctions.

Lors d'un appel de fonction, le compilateur recherche tout d'abord une correspondance exacte avec les fonctions « ordinaires ». S'il y a ambiguïté, la recherche échoue (comme à l'accoutumée). Si aucune fonction « ordinaire » ne convient, on examine alors tous les patrons ayant le nom voulu (en ne considérant que les paramètres de type). Si une seule correspondance exacte est trouvée, la fonction correspondante est instanciée (du moins, si elle ne l'a pas déjà été) et le problème est résolu. S'il y en a plusieurs, la recherche échoue.

Enfin, si aucun patron de fonction ne convient, on examine à nouveau toutes

les fonctions « ordinaires » en les traitant cette fois comme de simples fonctions surdéfinies (promotions numériques, conversions standard…).

Exercice 125

Énoncé

Créer un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque (le résultat possédera le même type). Écrire un petit programme utilisant ce patron.

Ici, notre patron ne comportera qu'un seul paramètre de type (correspondant à la fois à l'unique argument et à la valeur de retour de la fonction). Sa définition ne pose pas de problème particulier.

```
#include <iostream>
using namespace std ;
template <class T> T carre (T a)
  { return a * a ;
  }
main()
{ int n = 5 ;
  float x = 1.5 ;
  cout << "carre de " << n << " = " << carre (n) << "\n" ;
  cout << "carre de " << x << " = " << carre (x) << "\n" ;
}
```

Exercice 126

Énoncé

Soit cette définition de patron de fonctions :

```
template <class T, class U> T fct (T a, U b, T c)
{ .....
}
```

Avec les déclarations suivantes :

```
int n, p, q ;
float x ;
char t[20] ;
char c ;
```

Quels sont les appels corrects et, dans ce cas, quels sont les prototypes des fonctions instanciées ?

```
fct (n, p, q) ;          // appel I
fct (n, x, q) ;          // appel II
fct (x, n, q) ;          // appel III
fct (t, n, &c) ;         // appel IV
```

Solutions

a. L'appel I est correct ; il instancie la fonction :

```
int fct (int, int, int)
```

b. L'appel II est correct ; il instancie la fonction :

```
int fct (int, float, int)
```

c. L'appel III est incorrect.

d. L'appel IV est correct selon la norme. Il instancie la fonction :

```
char * fct (char *, int, char *)
```

Exercice 127

Énoncé

Créer un patron de fonctions permettant de calculer la somme d'un tableau d'éléments de type quelconque, le nombre d'éléments du tableau étant fourni en paramètre (on supposera que l'environnement utilisé accepte les « paramètres expression »). Écrire un petit programme utilisant ce patron.

Solution

```
// définition du patron de fonctions
template <class T> T somme (T * tab, int nelem)
{ T som ;
  int i ;
  som = 0 ;
  for (i=0 ; i<nelem ; i++)  som = som + tab[i] ;
  return som ;
}

// exemple d'utilisation
#include <iostream>
using namespace std ;
main()
{ int ti[] = {3, 5, 2, 1} ;
  float tf [] = {2.5, 3.2, 1.8} ;
  char tc[] = { 'a', 'e', 'i', 'o', 'u' } ;
  cout << somme (ti, 4) << "\n"  ;
  cout << somme (tf, 3) << "\n"  ;
  cout << somme (tc, 5) << "\n"  ;
}
```

Remarques

1. tel qu'il a été conçu, le patron `somme` ne peut être appliqué qu'à un type `T` pour lequel :

– l'opération d'addition a un sens ; cela signifie donc qu'il ne peut pas s'agir d'un type pointeur ; il peut s'agir d'un type classe, à condition que cette dernière ait surdéfini l'opérateur d'addition ;

– la déclaration `T som` est correcte ; cela signifie que si `T` est un type classe, il est nécessaire qu'il dispose d'un constructeur sans argument ;

– l'affectation `som = 0` est correcte ; cela signifie que si `T` est un type classe, il est nécessaire qu'il ait surdéfini l'affectation.

À ce propos, notons qu'il est possible d'initialiser `som` lors de sa déclaration, en procédant ainsi :

```
T som (0) ;
```

Cela est équivalent à `T som = 0` si `T` est un type prédéfini. En revanche,

si `T` est de type classe, cela provoque l'appel d'un constructeur à 1 argument de `T`, en lui transmettant la valeur 0 ; le problème relatif à l'affectation `som = 0` ne se pose plus alors.

2. L'exécution de l'exemple proposé fourni des résultats peu satisfaisants dans le cas où l'on applique `somme` à un tableau de caractères, compte tenu de la capacité limitée de ce type. On pourrait améliorer la situation en « spécialisant » notre patron pour les tableaux de caractères (en prévoyant, par exemple, une valeur de retour de type `int`).

Exercice 128

Énoncé

Soient les définitions suivantes de patrons de fonctions :

```
template <class T, class U> void fct (T a, U b)   { ... } //
patron I
template <class T, class U> void fct (T * a, U b) { ... } //
patron II
template <class T>          void fct (T, T, T)    { ... } //
patron III
void fct (int a, float b) { .....}                        //
fonction IV
```

Avec ces déclarations :

```
int n, p, q ;
float x, y ;
double z ;
```

Quels sont les appels corrects et, dans ce cas, quels sont les patrons utilisés et les prototypes des fonctions instanciées ?

```
fct (n, p) ;     // appel I
fct (x, y ) ;    // appel II
fct (n, x) ;     // appel III
fct (n, z) ;     // appel IV
fct (&n, p) ;    // appel V
fct (&n, x) ;    // appel VI
fct (&n, &p, &q) // appel VII
```

Solution

Ici, on fait appel à la fois à une surdéfinition (patrons I, II et III) et à une spécialisation de patron (fonction IV).

```
I)      patron I       void fct (int, int) ;
II)     patron I       void fct (float, float) ;
III)    fonction IV    void fct (int, float) ;
IV)     patron I       void fct (int, double) ;
V)      erreur : ambiguïté entre fct (T, U) et fct (T*, U)
VI)     erreur : ambiguïté entre fct (T, U) et fct (T*, U)
VII)    patron III     void fct (int *, int *, int *) ;
```

Remarque

Le patron II ne peut jamais être utilisé. En effet, chaque fois qu'il pourrait l'être, le patron I peut l'être également, de sorte qu'il y a ambiguïté. Le patron II est donc, ici, parfaitement inutile.

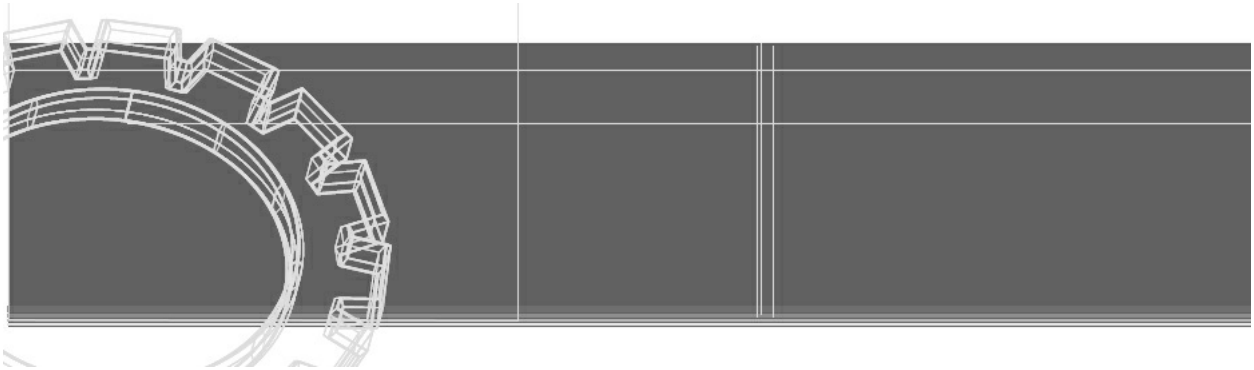
Notez que si nous avions défini simultanément les deux patrons :

```
template <class T, class U> void fct (T a, U b) ;
template <class T>          void fct (T a, T b)
```

le même phénomène d'ambiguïté (entre ces deux patrons) serait apparu lors d'appels tels que `fct (n,p)` ou `fct(x,y).`

Rappelons que l'ambiguïté n'est détectée que lorsque le compilateur doit instancier une fonction et non simplement au vu des définitions de patrons elles-mêmes : ces dernières restent donc acceptées tant que l'ambiguïté n'est pas mise en évidence par un appel la révélant.

Chapitre 18
Les patrons de classes



Rappels

Introduite par la version 3, la notion de patron de classes permet de définir ce que l'on nomme aussi des « classes génériques ». Plus précisément, à l'aide d'une seule définition comportant des paramètres de type et des paramètres expression, on décrit toute une famille de classes ; le compilateur fabrique (instancie) la ou les classes nécessaires à la demande (on nomme souvent ces instances des « classes patron »).

Remarque

Cette fois, les paramètres expression étaient déjà prévus par la version 3 alors que, dans le cas des patrons de fonctions, ils n'ont été introduits que par la norme ANSI.

Définition d'un patron de classes

On précise les paramètres de type en les faisant précéder du mot clé `class` et les paramètres expression en mentionnant leur type dans une liste de paramètres introduite par le mot `template` (comme pour les patrons de fonctions, avec cette différence qu'ici, tous les paramètres – type ou expression – apparaissent).

Par exemple :

```
template <class T, class U, int n> class gene
{ // ici, T désigne un type quelconque, n une valeur entière
quelconque
} ;
```

Si une fonction membre est définie (ce qui est le cas usuel) à l'extérieur de la définition du patron, il faut rappeler au compilateur la liste de paramètres (`template`) et préfixer l'en-tête de la fonction membre du nom du patron accompagné de ses paramètres. (En toute rigueur, il s'agit d'une redondance constatée, mais non justifiée, par le fondateur du langage lui-même, Stroustrup.) Par exemple, pour un constructeur de notre patron de classes précédent :

```
template <class T, class U, int n> gene <T, U, n>::gene (...)
{ ..... }
```

Instanciation d'une classe patron

On déclare une classe patron en fournissant à la suite du nom de patron un nombre de paramètres effectifs (noms de types ou expressions) correspondant aux paramètres figurant dans la liste (`template`). Les paramètres expression doivent obligatoirement être des expressions constantes du même type que celui figurant dans la liste. Par exemple, avec notre précédent patron (on suppose que `pt` est une classe) :

```
class gene <int, float, 5> c1 ;      // T = int, U = float,  n
= 5
class gene <int, int, 12>  c2 ;      // T = int, U = int,    n
= 12
const int NV=100 ;
class gene <pt, double, NV> c3 ;      // T = pt,  U = double,
n=100
int n = 5 ;
```

```
class gene <int, double, n> c4 ;      // erreur : n n'est pas
constant
const char C = 'e' ;
class gene <int, double, C> c5 ;      // erreur : C de type
char et non int
```

Un paramètre de type effectif peut lui-même être une classe patron. Par exemple, si nous avons défini un patron de classes `point` par :

```
template <class T> class point { ..... } ;
```

Voici des instances possibles de `gene` :

```
class gene <point<int>, float, 10> c5 ;                //
T=point<int>, U=float, n=10
class gene <point<char>, point<float>, 5> c6 ;  //
T=point<int>, U=point<float>, n=5
```

Un patron de classes peut comporter des membres (données ou fonctions) statiques ; dans ce cas, chaque instance de la classe dispose de son propre jeu de membres statiques.

Spécialisation d'un patron de classes

Un patron de classes ne peut pas être surdéfini (on ne peut pas définir deux patrons de même nom). En revanche, on peut spécialiser un patron de classes de différentes manières.

-- En spécialisant une fonction membre

Par exemple, avec ce patron :

```
template <class T, int n> class tableau { ..... } ;
```

On pourra écrire une version spécialisée de constructeur pour le cas où `T` est le type `point` et où `n` vaut `10` en procédant ainsi :

```
tableau <point, 10>:: tableau (...) { ..... }
```

-- En spécialisant une classe

Dans ce cas, on peut éventuellement spécialiser tout ou une partie des fonctions membre, mais ce n'est pas nécessaire. Par exemple, avec ce patron :

```
template <class T> class point { ..... } ;
```

on peut fournir une version spécialisée pour le cas où `T` est le type `char` en procédant ainsi :

```
class point <char>
{ // nouvelle définition de la classe point pour les caractères
} ;
```

Identité de classes patron

On ne peut affecter entre eux que deux objets de même type. Dans le cas d'objets d'un type classe `patron`, on considère qu'il y a identité de type lorsque leurs paramètres de types sont identiques et que les paramètres expression ont les mêmes valeurs.

Classes patron et héritage

On peut « combiner » de plusieurs façons l'héritage avec la notion de patron de classes :

- **Classe « ordinaire » dérivée d'une classe patron** ; par exemple, si `A` est une classe patron définie par `template <class T> A` :

  ```
  class B : public A <int>    // B dérive de la classe patron A<int>
  ```

  On obtient une seule classe nommée `B`.

- **Patron de classes dérivé d'une classe « ordinaire »** ; par exemple, si `A` est une classe ordinaire :

  ```
  template <class T> class B : public A
  ```

  On obtient une famille de classes (de paramètre de type `T`).

- **Patron de classes dérivé d'un patron de classes** ; par exemple, si `A` est une classe patron définie par `template <class T> A`, on peut :

  – définir une nouvelle famille de fonctions dérivées par :

  ```
  template <class T> class B : public A <T>
  ```

  Dans ce cas, il existe autant de classes dérivées possibles que de classes de base possibles.

– définir une nouvelle famille de fonctions dérivées par :

```
template <class T, class U> class B : public A <T>
```

Dans ce cas, on peut dire que chaque classe de base possible peut engendrer une famille de classes dérivées (de paramètre de type `U`).

Exercice 129

Énoncé

Soit la définition suivante d'un patron de classes :

```
template <class T, int n> class essai
{  T tab [n] ;
  public :
   essai (T) ;    // constructeur
} ;
```

a. Donnez la définition du constructeur `essai`, en supposant :

- qu'elle est fournie « l'extérieur » de la définition précédente ;
- que le constructeur recopie la valeur reçue en argument dans chacun des éléments du tableau `tab` .

b. Disposant ainsi de la définition précédente du patron `essai`, de son constructeur et de ces déclarations :

```
const int n = 3 ;
int p = 5 ;
```

Quelles sont les instructions correctes et les classes instanciées ? On en fournira (dans chaque cas) une définition équivalente sous la forme d'une « classe ordinaire », c'est-à-dire dans laquelle la notion de paramètre a disparu.

```
essai <int, 10> ei (3) ;              // I
essai <float, n> ef (0.0) ;           // II
essai <double, p> ed (2.5) ;          // III
```

Solutions

a. La définition du constructeur est analogue à celle que l'on aurait écrite « en ligne » ; il faut simplement « préfixer » son en-tête d'une liste de paramètres introduite par `template`. De plus, il faut préfixer l'en-tête de la fonction membre du nom du patron accompagné de ses paramètres (bien que cela soit redondant) :

```
template <class T, int n> essai<T,n>::essai(T a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b. Appel I : correct.

```
class essai
{  int tab [10] ;
  public :
   essai (int) ;    // constructeur
} ;

essai::essai (int a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b. Appel II : correct.

```
class essai
{  float tab [n] ;
  public :
   essai (float) ;  // constructeur
} ;

essai::essai (float a)
{  int i ;
   for (i=0 ; i<n ; i++) tab[i] = a ;
}
```

b. Appel III : incorrect, car `p` n'est pas une expression constante.

Exercice 130

Énoncé

a. Créer un patron de classes nommé `pointcol`, tel que chaque classe instanciée permette de manipuler des points colorés (deux coordonnées et une couleur) pour lesquels on puisse « choisir » à la fois le type des coordonnées et celui de la couleur. On se limitera à deux fonctions membre : un constructeur possédant trois arguments (sans valeur par défaut) et une fonction `affiche` affichant les coordonnées et la couleur d'un « point coloré ».

b. Dans quelles conditions peut-on instancier une classe patron `pointcol` pour des paramètres de type classe ?

Solutions

a. Voici ce que pourrait être la définition du patron demandé, en prévoyant les fonctions membre « en ligne » :

```
template <class T, class U> class pointcol
{
   T x, y ;    // coordonnées
   U coul ;    // couleur
  public :
   pointcol (T abs, T ord, U cl)
     { x = abs ; y = ord ; coul = cl ;
     }
   void affiche ()
     { cout << "point coloré - coordonnées " << x << " " <<
y
            << " couleur " << coul << "\n" ;
     }
} ;
```

À titre indicatif, voici un exemple d'utilisation (on suppose que la définition précédente figure dans `pointcol.h`) :

```
#include "pointcol.h"
#include <iostream>
using namespace std ;

main()
{ pointcol <int, short int > p1 (5, 5, 2) ;  p1.affiche () ;
  pointcol <float, int> p2 (4, 6, 2) ; p2.affiche () ;
  pointcol <double, unsigned short> p3 (1, 5, 2) ; p3.affiche
() ;
}
```

b. Il suffit que le type classe en question ait convenablement surdéfini l'opérateur <<, afin d'assurer convenablement l'affichage sur `cout` des informations correspondantes.

Exercice 131

Énoncé

On a défini le patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnées
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
     { cout << "Coordonnées : " << x << " " << y << "\n" ;
     }
```

a. Que se passe-t-il avec ces instructions :

```
point <char> p (60, 65) ;
p.affiche () ;
```

b. Comment faut-il modifier la définition de notre patron pour que les instructions précédentes affichent bien :

```
Coordonnées : 60 65
```

Solutions

a. On obtient l'affichage des caractères de code `60` et `65` (c'est-à-dire dans une implémentation utilisant le code ASCII : < et `A`) et non les nombres `60` et `65`.

b. Il faut spécialiser notre patron `point` pour le cas où le type `T` est le type `char`. Pour ce faire, on peut :

- soit fournir une définition complète de `point<char>`, avec ses fonctions membre ;

- soit, puisqu'ici seule la fonction `affiche` est concernée, se contenter de surdéfinir la fonction `point<char>::affiche`, ce qui conduit à cette nouvelle définition de notre patron :

```
            // définition générale du patron point
template <class T> class point
{
   T x, y ;  // coordonnées
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{ cout << "Coordonnées : " << x << " " << y << "\n" ;
}
    // version spécialisée de la fonction affiche pour le type
char
void point<char>::affiche ()
{ cout << "Coordonnées : " << (int)x << " " << (int)y <<
"\n" ;
}
```

Exercice 132

Énoncé

Créer un patron de classes permettant de représenter des « vecteurs dynamiques » c'est-à-dire des vecteurs dont la dimension peut ne pas être connue lors de la compilation (ce n'est donc pas obligatoirement une expression constante comme dans le cas de tableaux usuels). On prévoira que les éléments de ces vecteurs puissent être de type quelconque.

On surdéfinira convenablement l'opérateur `[]` pour qu'il permette l'accès aux éléments du vecteur (aussi bien en consultation qu'en modification) et on s'arrangera pour qu'il n'existe aucun risque de « débordement d'indice ». En revanche, on ne cherchera pas à régler les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets du type concerné.

N.B. Il ne faut pas chercher à utiliser les composants standard introduits

par la norme. En effet, le patron `vector` répondrait intégralement à la question.

Solution

En généralisant ce qui a été fait dans l'exercice 92 (sans toutefois initialiser les éléments du vecteur lors de sa construction), nous aboutissons au patron de classes suivant :

```
template <class T> class vect
{  int nelem ;    // nombre d'éléments
    T * adr ;        // adresse zone dynamique contenant les
éléments
  public :
   vect (int) ;              // constructeur
   ~vect () ;                // destructeur
   T & operator [] (int) ;   // opérateur d'accès à un élément
} ;
template <class T> vect<T>::vect (int n)
{ adr = new T [nelem = n] ;
}
template <class T> vect<T>::~vect ()
{ delete [] adr ;
}
template <class T> T & vect<T>::operator [] (int i)
{ if ( (i<0) || (i>nelem) ) i = 0 ;     // protection indice
hors limites
  return adr [i] ;
}
```

Remarque

La définition du patron de classes serait plus simple si les fonctions membre étaient « en ligne ».

Notez que, ici encore, nous avons fait en sorte qu'une tentative d'accès à un élément situé en dehors du vecteur conduise à accéder à l'élément de rang `0`. Dans la pratique, on aura intérêt à utiliser des protections plus élaborées.

À titre indicatif, voici un petit programme, accompagné du résultat fourni par son exécution, utilisant ce patron (dont on suppose que la définition figure

dans `vectgen.h`) :

```
#include "vectgen.h"
#include <iostream>
using namespace std ;
main()
{ vect<int> vi (10) ;
  vi[5] = 5 ; vi[2] = 2 ;
  cout << vi[2] << " " << vi[5] << "\n" ;
  vect<double> vd (3) ;
  vd[0] = 0.0 ; vd[1] = 0.1 ; vd[2] = 0.2 ;
  cout << vd[0] << " " << vd[1] << " " << vd[2] << "\n" ;
  cout << vd[12] << "\n" ;
  vd[12] = 1.2 ; cout << vd[12] << " " << vd[0] ;
}
```

```
2 5
0 0.1 0.2
0
1.2 1.2
```

Notre patron `vect` permet d'instancier des vecteurs dynamiques dans lesquels les éléments sont de type absolument quelconque, en particulier de type classe (pourvu que ladite classe dispose d'un constructeur sans argument). Il n'en serait pas allé ainsi si nous avions initialisé les éléments du tableau lors de leur construction par :

```
int i ;
for (i=0 ; i<nelem ; i++) adr[i] = 0 ;
```

En effet, dans ce cas, ces instructions auraient convenablement fonctionné pour n'importe quel type de base (par conversion de l'entier `0` dans le type voulu). En revanche, pour être applicable à des éléments de type `classe`, il aurait fallu, en outre, que la classe concernée dispose d'une conversion d'un `int` dans ce type classe, c'est-à-dire d'un constructeur à un argument de type numérique.

Exercice 133

Énoncé

Comme dans l'exercice précédent, réaliser un patron de classes permettant de manipuler des vecteurs dont les éléments sont de type quelconque mais pour lesquels la dimension, supposée être cette fois une expression constante, apparaîtra comme un paramètre (expression) du patron. Hormis cette différence, les « fonctionnalités » du patron resteront les mêmes.

Solution

Il n'est plus nécessaire d'allouer un emplacement dynamique pour notre vecteur, qui peut donc figurer directement dans les membres donnée de notre patron. Le constructeur n'est plus nécessaire (voir toutefois la remarque ci-dessous), pas plus que le destructeur. Voici ce que pourrait être la définition de notre patron :

```
template <class T, int n> class vect
{  T v [n] ;      // vecteur de n éléments de type T
  public :
   T & operator [] (int) ;   // opérateur d'accès à un élément
} ;
template <class T, int n> T & vect<T,n>::operator [] (int i)
{ if ( (i<0) || (i>n) ) i = 0 ;     // protection indice hors
limites
  return v [i] ;
}
```

Voici, toujours à titre indicatif, ce que deviendrait le petit programme d'essai :

```
#include "vectgen1.h"
#include <iostream>
using namespace std ;
main()
{ vect<int, 10> vi  ;
  vi[5] = 5 ; vi[2] = 2 ;
  cout << vi[2] << " " << vi[5] << "\n" ;
  vect<double, 3> vd ;
  vd[0] = 0.0 ; vd[1] = 0.1 ; vd[2] = 0.2 ;
```

```
   cout << vd[0] << " " << vd[1] << " " << vd[2] << "\n" ;
   cout << vd[12] << "\n" ;
   vd[12] = 1.2 ; cout << vd[12] << " " << vd[0] ;
 }
```

Remarque

Ici, nous n'avons pas eu besoin de faire du nombre d'éléments un membre donnée de nos classes patron : en effet, lorsqu'on en a besoin, on l'obtient comme étant la valeur du second paramètre fourni lors de l'instanciation. Si nous avions voulu conserver ce nombre d'éléments sous forme d'un membre donnée, il aurait été nécessaire de prévoir un constructeur, par exemple :

```
template <class T, int n> class vect
{  int nelem ;    // nombre d'éléments
   T v [n] ;      // vecteur de n éléments de type T
  public :
   vect () ;
    T & operator [] (int) ;    // opérateur d'accès à un
élément
} ;
template <class T, int n> vect<T,n>::vect ()
{ nelem = n ;
}
```

Exercice 134

Énoncé

On dispose du patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnées
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnées : " << x << " " << y << "\n" ;
     }
} ;
```

a. Créer, par dérivation, un patron de classes `pointcol` permettant de manipuler des « points colorés » dans lesquels les coordonnées et la couleur sont de même type. On redéfinira convenablement les fonctions membre en réutilisant les fonctions membre de la classe de base.

b. Même question, mais en prévoyant que les coordonnées et la couleur puissent être de deux types différents.

c. Toujours par dérivation, créer cette fois une « classe ordinaire » (c'est-à-dire une classe qui ne soit plus un patron de classes, autrement dit qui ne dépende plus de paramètres...) dans laquelle les coordonnées sont de type `int`, tandis que la couleur est de type `short`.

Solutions

a. Aucun problème particulier ne se pose ; il suffit de faire dériver `pointcol<T>` de `point<T>`. Voici ce que peut être la définition de notre patron (ici, nous avons laissé le constructeur « en ligne » mais nous avons défini `affiche` en dehors de la classe) :

```
template <class T> class pointcol : public point<T>
{  T cl ;
  public :
   pointcol (T abs, T ord, T coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche () ;
} ;

template <class T> void pointcol<T>::affiche ()
{  point<T>::affiche () ;
   cout << "    couleur : " << cl << "\n" ;
}
```

Voici un petit exemple d'utilisation (il nécessite les déclarations appropriées ou l'incorporation des fichiers en-tête correspondants) :

```
main()
{ pointcol <int> p1 (2, 5, 1) ; p1.affiche () ;
  pointcol <float> p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

b. Cette fois, la classe dérivée dépend de deux paramètres (nommés ici `T` et

U). Voici ce que pourrait être la définition de notre patron (avec, toujours, un constructeur en ligne et une fonction `affiche` définie à l'extérieur de la classe) :

```
template <class T, class U> class pointcol : public point<T>
{  U cl ;
  public :
   pointcol (T abs, T ord, U coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche () ;
} ;

template <class T, class U> void pointcol<T, U>::affiche ()
{  point<T>::affiche () ;
   cout << "    couleur : " << cl << "\n" ;
}
```

Voici un exemple d'utilisation (on suppose qu'il est muni des déclarations appropriées) :

```
main()
{
  pointcol <int, short> p1 (2, 5, 1) ; p1.affiche () ;
  pointcol <float, int> p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

c. Cette fois, `pointcol` est une simple classe, ne dépendant plus d'aucun paramètre. Voici ce que pourrait être sa définition :

```
class pointcol : public point<int>
{
   short cl ;
  public :
    pointcol (int abs, int ord, short coul) : point<int>
(abs, ord)
     { cl = coul ;
     }
   void affiche ()
     { point<int>::affiche () ;
       cout << "    couleur : " << cl << "\n" ;
     }
} ;
```

Et un petit exemple d'utilisation :

```
main()
{
```

```
  pointcol  p1 (2, 5, 1) ; p1.affiche () ;
  pointcol  p2 (2.5, 5.25, 4) ; p2.affiche () ;
}
```

Exercice 135

Énoncé

On dispose du même patron de classes que précédemment :

```
template <class T> class point
{
   T x, y ;    // coordonnées
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
   void affiche ()
     { cout << "Coordonnées : " << x << " " << y << "\n" ;
     }
} ;
```

a. Lui ajouter une version spécialisée de `affiche` pour le cas où `T` est le type caractère.

b. Comme dans la question **a** de l'exercice précédent, créer un patron de classes `pointcol` permettant de manipuler des « points colorés » dans lesquels les coordonnées et la couleur sont de même type. On redéfinira convenablement les fonctions membre en réutilisant les fonctions membre de la classe de base et l'on prévoira une version spécialisée de `affiche` de `pointcol` dans le cas du type caractère.

Solutions

a.

```
void point<char>::affiche ()
{ cout << "Coordonnées : " << (int)x << " " << (int)y <<
"\n" ;
```

b.

```
template <class T> class pointcol : public point<T>
```

```
{  T cl ;
  public :
   pointcol (T abs, T ord, T coul) : point<T> (abs, ord)
     { cl = coul ;
     }
   void affiche ()
    {  point<T>::affiche () ;
       cout << " couleur : " << cl << "\n" ;
    }
} ;
void pointcol<char>::affiche ()
{  point<char>::affiche () ;
   cout << " couleur : " << (int)cl << "\n" ;
}
}
```

Remarque

Seule la question **a** de l'exercice 134 se prêtait à une spécialisation pour le type caractère. En effet, pour la classe demandée en **c**, n'ayant plus affaire à un patron de classes, la question n'aurait aucun sens. En ce qui concerne la classe demandée en **b**, en revanche, on se trouve en présence d'une classe dérivée dépendant de 2 paramètres `T` et `U`. Il faudrait alors pouvoir spécialiser une classe, non plus pour des valeurs données de tous les (deux) paramètres, mais pour une valeur donnée (`char`) de l'un d'entre eux ; cette notion de famille de spécialisation n'existe pas dans la norme actuelle de C++.

Exercice 136

Énoncé

On dispose du patron de classes suivant :

```
template <class T> class point
{  T x, y ;    // coordonnées
  public :
   point (T abs, T ord) { x = abs ; y = ord ; }
```

```
    void affiche ()
      { cout << "Coordonnées : " << x << " " << y << "\n" ;
      }
} ;
```

On souhaite créer un patron de classes `cercle` permettant de manipuler des cercles, définis par leur centre (de type `point`) et un rayon. On n'y prévoira, comme fonctions membre, qu'un constructeur et une fonction `affiche` se contentant d'afficher les coordonnées du centre et la valeur du rayon.

a. Le faire par héritage (un cercle est un point qui possède un rayon).

b. Le faire par composition d'objets membre (un cercle possède un point et un rayon).

Solutions

a. La démarche est analogue à celle de la question **a** de l'exercice 134. On a affaire à un patron dépendant de deux paramètres (ici `T` et `U`) :

```
template <class T, class U> class cercle : public point<T>
{  U r ;       // rayon
  public :
   cercle (T abs, T ord, U ray) : point<T> (abs, ord)
     { r = ray ;
     }
   void affiche ()
     { point<T>::affiche () ;
       cout << "    rayon : " << r ;
     }
} ;
```

b. Le patron dépend toujours de deux paramètres (`T` et `U`) mais il n'y a plus de notion d'héritage :

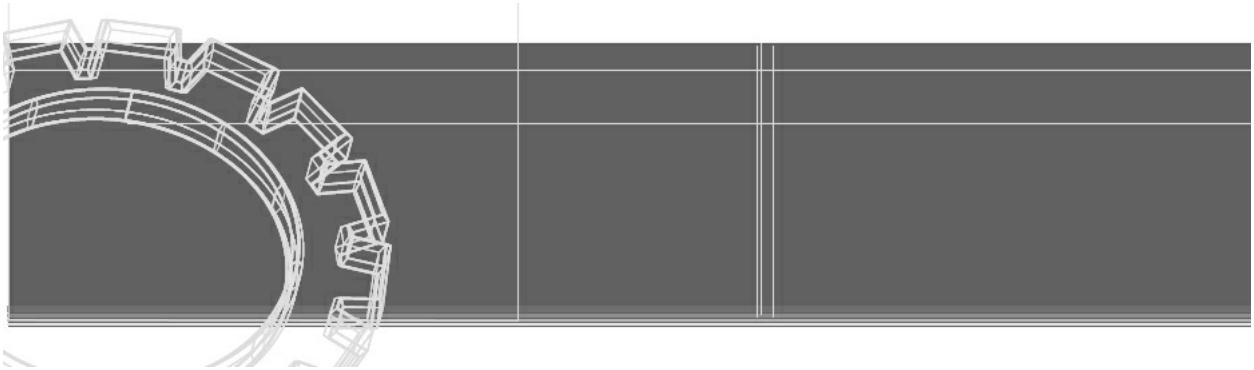
```
template <class T, class U> class cercle
{  point<T> c ;  // centre
   U r ;         // rayon
  public :
    cercle (T abs, T ord, U ray) : c(abs, ord)       //
pourrait r(ray)
     { r = ray ;
     }
   void affiche ()
```

```
        { c.affiche () ;
          cout << "    rayon : " << r ;
        }
};
```

Notez que, dans la définition du constructeur `cercle`, nous avons transmis les arguments `abs` et `ord` à un constructeur de `point` pour le membre `c`. Nous aurions pu utiliser la même notation pour `r`, bien que ce membre soit d'un type de base et non d'un type classe ; cela nous aurait conduit au constructeur suivant (de corps vide) :

```
cercle (T abs, T ord, U ray) : c(abs, ord), r(ray)
  { }
```

Chapitre 19
Gestion des exceptions



Rappels

Le mécanisme général

C++ dispose d'un mécanisme dit « de gestion des exceptions ». Une exception est une rupture de séquence (pas un appel de fonction !) déclenchée (on dit aussi « levée ») par un programme à l'aide de l'instruction `throw` dans laquelle on mentionne une expression quelconque. Il y a alors branchement à un ensemble d'instructions, dit « gestionnaire d'exceptions », choisi en fonction de la nature de l'expression indiquée à `throw`.

Pour qu'une portion de programme puisse intercepter une exception, il est nécessaire qu'elle figure à l'intérieur d'un bloc précédé du mot-clé `try`. Ce dernier soit être suivi d'une ou de plusieurs instructions `catch` représentant les différents gestionnaires correspondants, comme dans ce schéma :

```
try
{ .....  // instructions susceptibles de lever une exception,
soit
```

```
          // directement par throw (exp), soit par le biais de
fonctions
         // appelées
}
catch (type_a ...)
{ .....   // traitement de l'exception correspondant au type
type_a
}
catch (type_b ...)
{ .....   // traitement de l'exception correspondant au type
type_b
}
.....
catch (type_n ...)
{ .....   // traitement de l'exception correspondant au type
type_n
}
```

Un gestionnaire d'exceptions peut contenir des instructions `exit` ou `abort` qui mettent fin à l'exécution du programme. Une instruction `return` fait sortir de la fonction ayant levé l'exception. Dans les autres cas (rarement utilisés), on passe aux instructions suivant le bloc `try` concerné.

Algorithme de choix d'un gestionnaire d'exceptions

Lorsqu'une exception est levée par `throw`, avec le type `T`, on recherche, dans cet ordre, un gestionnaire correspondant à l'un des types suivants : type `T`, type de base de `T`, pointeur sur une classe dérivée (si `T` est d'un type pointeur sur une classe), type indéterminé (indiqué par `catch(...))` dans le gestionnaire.

Cheminement des exceptions

Lorsqu'une exception est levée par une fonction, on cherche tout d'abord un gestionnaire dans l'éventuel bloc `try` associé à cette fonction ; si l'on n'en trouve pas (ou si aucun bloc `try` n'est associé), on poursuit la recherche dans un éventuel bloc `try` associé à une fonction appelante et ainsi de suite. Si aucun gestionnaire d'exceptions n'est trouvé, on appelle la fonction `terminate` qui, par défaut appelle `abort`, laquelle fournit un message du genre `abnormal program termination`.

Spécification d'interface

Une fonction (y compris `main`) peut spécifier les exceptions qu'elle est susceptible de provoquer (elle-même, ou dans les fonctions qu'elle appelle à son tour). Elle le fait à l'aide du mot-clé `throw`, suivi, entre parenthèses, de la liste des exceptions concernées. Dans ce cas, toute exception non prévue et levée à l'intérieur de la fonction (ou d'une fonction appelée) entraîne l'appel d'une fonction particulière nommée `unexpected`. Par défaut, cette fonction appelle la fonction `terminate`.

Exercice 137

Énoncé

Quels résultats fournira ce programme lorsqu'on lui fournit comme données :

a. la valeur `1` ?

b. la valeur `0` ?

```
#include <iostream>
using namespace std ;
main()
{ int n ;
  cout << "donnez un entier : " ;
  cin >> n ;
  try
  { cout << "début premier bloc try\n" ;
    if (n) throw n ;
    cout << "fin premier bloc try\n" ;
  }
  catch (int n)
  { cout << "catch 1 - n = " << n << "\n" ;
  }

  cout << "suite programme\n" ;
  try
  { cout << "début second bloc try\n" ;
    throw n ;
    cout << "fin second bloc try\n" ;
```

```
    }
    catch (int n)
    { cout << "catch 2 - n = " << n << "\n" ;
    }
    cout << "fin programme\n" ;
  }
```

Solutions

a. Avec la valeur `1` :

```
donnez un entier : 1
debut premier bloc try
catch 1 - n = 1      // fourni par le bloc catch(int) associé
au premier
                     // bloc try
suite programme
debut second bloc try
catch 2 - n = 1   // fourni par le bloc catch(int) associé au
second bloc
                  // try
fin programme
```

On notera bien que, dans le premier bloc `try`, l'exception de type `int` provoque un branchement au bloc `catch(int)` correspondant. Comme ce dernier ne comporte pas d'instruction d'interruption de programme ou de fonction, on passe aux instructions suivant le dernier gestionnaire associé, c'est-à-dire ici au bloc `try` suivant. Là encore, une exception de type `int` provoque le branchement au bloc `catch(int)` correspondant (différent du précédent). Puis l'on passe aux instructions suivantes, c'est-à-dire ici à l'instruction affichant `fin programme`.

b. Avec la valeur `0` :

```
donnez un entier : 0
debut premier bloc try
fin premier bloc try
suite programme
debut second bloc try
catch 2 - n = 0
fin programme
```

Ici, contrairement à ce qui se produisait dans l'exécution précédente, le premier bloc `try` ne déclenche plus d'exception. Son exécution se poursuit donc en entier, d'où le message `fin premier bloc try`. La suite est

identique.

Exercice 138

Énoncé

Quels résultats donne ce programme lorsqu'on lui fournit comme données :

a. la valeur `1` ?

b. la valeur `2` ?

c. la valeur `3` ?

d. la valeur `4` ?

```
#include <stdlib.h>  // pour exit
#include <iostream>
using namespace std ;
main()
{ int n ; float x ; double z ;
  cout << "donnez un entier : " ;
  cin >> n ;
  try
  { switch (n)
   { case 1 : throw n ; break ;
     case 2 : x = n ; throw x ; break ;
     case 3 : z = n ; throw z ; break ;
    }
  }
  catch (int n)
  { cout << "catch entier - n = " << n << "\n" ;
  }
  catch (float x)
  { cout << "catch flottant - x = " << x << "\n" ;
    exit (-1) ;
  }
  cout << "suite et fin du programme\n" ;
}
```

Solutions

a.

```
donnez un entier : 1
catch entier - n = 1
suite et fin du programme
```

L'exception de type `int` levée dans le bloc `try` est interceptée par le gestionnaire `catch(int)` qui affiche un message. On passe alors à l'instruction suivant le bloc `try`, qui affiche le message `suite et fin du programme`.

b.

```
donnez un entier : 2
catch flottant - x = 2
```

L'exception de type `float` levée dans le bloc `try` est interceptée par le gestionnaire `catch(float)` qui affiche un message avant de mettre fin à l'exécution du programme.

c.

```
donnez un entier : 3
abnormal program termination      /* ou quelque chose de
semblable */
```

L'exception de type `double` levée dans le bloc `try` ne dispose d'aucun gestionnaire approprié (il aurait dû être du type `catch(double)`. On appelle donc la fonction `terminate` qui, par défaut, appelle la fonction `abort`, laquelle met fin à l'exécution du programme en affichant un message approprié.

d.

```
donnez un entier : 4
suite et fin du programme
```

Ici, aucune exception n'a été levée par le bloc `try` et on exécute l'instruction qui le suit, laquelle affiche le message de fin de programme.

Exercice 139

Énoncé

Quels résultats produira ce programme ?

```
#include <iostream>
using namespace std ;
class erreur
{ public :
   int num ;
} ;
class erreur_d : public erreur
{ public :
    int code ;
} ;
class A
{ public :
   A(int n)
   {if (n==1) { erreur_d erd ; erd.num = 999 ;
                erd.code = 12 ; throw erd ; }
   }
} ;

main()
{
  try
  { A a(1) ;
    cout << "après création a(1)\n" ;
  }
  catch (erreur er)
  { cout << "exception erreur : " << er.num << "\n" ;
  }
  catch (erreur_d erd)
   { cout << "exception erreur_d : " << erd.num << " " <<
erd.code << "\n" ;
  }

  cout << "suite\n" ;
  try
  { A b(1) ;
    cout << "après création b(1)\n" ;
  }

  catch (erreur_d erd)
   { cout << "exception erreur_d : " << erd.num << " " <<
erd.code << "\n" ;
  }
```

```
    catch (erreur er)
    { cout << "exception erreur : " << er.num << "\n" ;
    }
  }
```

Solution

```
exception erreur : 999
suite
exception erreur_d : 999 12
```

Dans le premier bloc `try`, la construction de l'objet `a(1)` lève une exception de type `erreur_d`, en transmettant une expression (`erd`) de ce type dans laquelle les champs `num` et `code` valent respectivement `999` et `12`. Le premier gestionnaire trouvé `(catch(erreur  er))` convient puisque `erreur` est un type de base de `erreur_d`. C'est donc lui qui est exécuté, ce qui explique que la valeur du champ `code` ne soit pas affichée, et ceci malgré l'existence, un peu plus loin, d'un gestionnaire mieux approprié `(catch(erreur_d))`.

Dans le second bloc `try`, en revanche, les mêmes gestionnaires ont été disposés dans l'ordre inverse. L'exception levée par la construction de `b` est alors traitée par le gestionnaire approprié et la valeur du champ `code` est effectivement affichée.

Exercice 140

Énoncé

Soit la définition suivante des classes `erreur` et `A` :

```
class erreur
{ public :
   int num ;
} ;
class A
{ public :
   A(int n)
   { if (n==1) { erreur er ; er.num = 999 ; throw er ; }
   }
```

```
} ;
```

1. Quels résultats fournira ce programme utilisant ces deux classes :

```
#include <iostream>
using namespace std ;
main()
{ void f() ;
  try
  { f() ;
  }
  catch (erreur er)
  { cout << "dans main : " << er.num << "\n" ;
  }
  cout << "suite main\n" ;
}
void f()
{ try
  { A a(1) ;
  }
  catch (erreur er)
  { cout << "dans f : " << er.num << "\n" ;
  }
  cout << "suite f\n" ;
}
```

2. Même question en remplaçant la définition de `f` par :

```
void f()
{ try
  { A a(1) ;
  }
  catch (erreur er)
  { cout << "dans f : " << er.num << "\n" ;
    return ;
  }
  cout << "suite f\n" ;
}
```

3. Même question en remplaçant la définition de `f` par :

```
void f()
{  A a(1) ;
}
```

Solutions

1.

```
dans f : 999
suite f
suite main
```

L'exception levée par la construction dans `f` de `a(1)` est traitée par le gestionnaire `catch(erreur)` associé au bloc `try` correspondant de la fonction elle-même. On exécute ensuite l'instruction suivant ce bloc, laquelle affiche `suite f`, avant d'effectuer un retour classique de la fonction `f` dans `main`. On notera que le bloc `try` de `main` n'intervient pas ici.

2.

```
dans f : 999
suite main
```

Là encore, l'exception levée `f` par la construction dans `f` de `a(1)` est traitée par le gestionnaire `catch(erreur)` associé au bloc `try` correspondant de la fonction elle-même. Mais cette fois, celui-ci se termine par une instruction `return`, laquelle provoque le retour de la fonction concernée, à savoir `f` (attention, pas le gestionnaire d'exceptions, qui n'est pas une fonction !).

3.

```
dans main : 999
suite main
```

Cette fois, la construction de `a(1)` dans `f` ne figure pas dans un bloc `try`. La recherche du gestionnaire de l'exception alors levée se fait dans un bloc `try` englobant, à savoir ici celui dans lequel a eu lieu l'appel de `f`.

Exercice 141

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
class A
{ public :
   A(int n)
```

```
     { try
       { if (n==1) throw n ;
       }
       catch (int n)
       { cout << "dans constructeur A : " << n << "\n" ;
         throw ;
       }
     }
  } ;
  main()
  { void f() ;
    try
    { f() ;
    }
    catch (int n)
    { cout << "dans main : " << n << "\n" ;
    }
    cout << "fin main\n" ;
  }
  void f()
  {
    try
    { A a(1) ;
    }
    catch (int n)
    { cout << "dans f : " << n << "\n" ;
      throw ;
    }
    cout << "suite f\n" ;
  }
```

Solution

```
dans constructeur A : 1
dans f : 1
dans main : 1
fin main
```

L'exception levée par la construction de l'objet `a(1)` est tout d'abord interceptée par le gestionnaire associé au bloc `try` du constructeur, d'où le premier message. Mais l'instruction `throw` qu'il contient demande de transmettre l'exception au bloc `try` englobant, à savoir ici celui figurant dans la fonction `f` d'où le second message. Là encore, une instruction `throw` retransmet l'exception au niveau supérieur, à savoir le bloc `try` de `main`.

Exercice 142

Énoncé

Écrire une classe `point` (à deux coordonnées entières) qui dispose d'un constructeur à deux arguments levant une exception lorsque les deux composantes sont égales. De plus, l'appel d'un constructeur sans argument ou à un seul argument devra également lever un autre type d'exception. Ici, on limitera les fonctions membre de `point` aux seuls constructeurs.

Écrire un programme d'utilisation de la classe `point`, interceptant convenablement les exceptions prévues, en mentionnant la cause.

Solution

Il est préférable d'associer un type classe à chacune des deux sortes d'exceptions prévues. Nous choisirons `er_compos` pour l'exception déclenchée en cas d'égalité des composantes et `er_constr` pour celle déclenchée en cas d'appel d'un constructeur à `0` ou `1` argument ; la distinction entre les deux se fera par une valeur entière transmise au constructeur et conservée ici dans un champ public. Voici ce que pourrait être la définition de notre classe :

```
class er_compos
{ } ;
class er_constr
{ public :
  int num ;
  er_constr (int n) { num = n ; }
} ;
class point
{  int x, y ;
  public :
   point (int abs, int ord)
   { if (abs==ord) throw new er_compos ;
     x=abs ; y=ord ;
   }
   point ()
   { throw new er_constr (0) ;
   }
```

```
   point (int abs)
   { throw new er_constr (1);
   }
} ;
```

Voici un exemple de programme qui se contente de signaler les exceptions interceptées en interrompant l'exécution. On notera que les trois constructions proposées provoquent effectivement une exception qui, au bout du compte, interrompt le programme. Autrement dit, pour percevoir l'effet de la seconde ou de la troisième, il faudrait la placer avant les autres.

```
#include <iostream>
using namespace std ;
main()
{
  try
  { // .....
       point b(1, 1) ;   // afficherait : exception création
point :
                       // composantes égales
      point () ;         // afficherait : exception création
point :
                       // constructeur 0 arg
      point (3) ;        // afficherait : exception création
point :
                       // constructeur 1 arg
     // .....
  }
  catch (er_compos e)
   { cout << "exception création point : composantes égales\n
" ;
    exit (-1) ;
  }
  catch (er_constr ec)
  { switch (ec.num)
       { case 1 :   cout << "exception création point :
constructeur 0 arg\n" ;
                break ;
          case 2 :   cout << "exception création point :
constructeur 1 arg\n" ;
                 break ;
    }
    exit(-1) ;
  }
}
```

Exercice 143

Énoncé

1. Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{ void f() ;
  try
  { f() ;
  }
  catch (int n)
  { cout << "except int dans main : " << n << "\n" ;
  }
  catch (...)
  { cout << "exception autre que int dans main \n" ;
  }
  cout << "fin main\n" ;
}
void f()
{
  try
  {  int n=1 ; throw n ;
  }
  catch (int n)
  { cout << "except int dans f : " << n << "\n" ;
    throw ;
  }
}
```

2. Même question si l'on remplace la fonction `f` par :

```
void f()
{
  try
  {  float x=2.5 ; throw x ;
  }
  catch (int n)
  { cout << "except int dans f : " << n << "\n" ;
    throw ;
  }
}
```

Solutions

1.

```
except int dans f : 1
except int dans main : 1
fin main
```

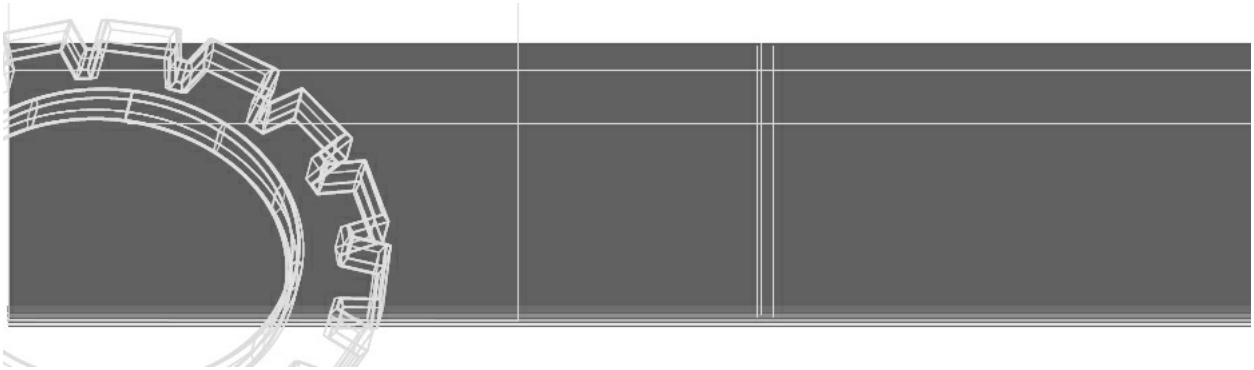
L'exception de type `int` levée dans `f` est traitée par le gestionnaire `catch(int)` correspondant. Elle est retransmise à un bloc englobant par `throw`, donc traitée par le gestionnaire `catch(int)` du bloc `try` du `main`.

2.

```
exception autre que int dans main
fin main
```

Cette fois, aucun gestionnaire approprié n'existe pour traiter l'exception de type `float` levée dans la fonction `f`. On recherche un gestionnaire approprié dans un bloc `try` englobant, ce qui conduit ici à `catch(...)`.

Chapitre 20
Exercices de synthèse



Exercice 144

Énoncé

Réaliser une classe nommée `set_int` permettant de manipuler des ensembles de nombres entiers. Le nombre maximal d'entiers que pourra contenir l'ensemble sera précisé au constructeur qui allouera dynamiquement l'espace nécessaire. On prévoira les opérateurs suivants (`e` désigne un élément de type `set_int` et `n` un entier :

- `<<`, tel que `e<<n` ajoute l'élément `n` à l'ensemble `e` ;
- `%`, tel que `n % e` vale `1` si `n` appartient à `e` et `0` sinon ;
- `<<`, tel que `flot << e` envoie le contenu de l'ensemble `e` sur le flot indiqué, sous la forme :

```
[entier1, entier2, ... entiern]
```

La fonction membre `cardinal` fournira le nombre d'éléments de

l'ensemble. Enfin, on s'arrangera pour que l'affectation ou la transmission par valeur d'objets de type `set_int` ne pose aucun problème (on acceptera la duplication complète d'objets).

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

Naturellement, notre classe comportera, en membres donnée, le nombre maximal (`nmax`) d'éléments de l'ensemble, le nombre courant d'éléments (`nelem`) et un pointeur sur l'emplacement contenant les valeurs de l'ensemble.

Comme notre classe comporte une partie dynamique, il est nécessaire, pour que l'affectation et la transmission par valeur se déroulent convenablement, de surdéfinir l'opérateur d'affectation et de munir notre classe d'un constructeur par recopie. Les deux fonctions membre (`operator =` et `set_int`) devront prévoir une « copie profonde » des objets. Nous utiliserons pour cela une méthode que nous avons déjà rencontrée et qui consiste à considérer que deux objets différents disposent systématiquement de deux parties dynamiques différentes, même si elles possèdent le même contenu.

L'opérateur `%` doit être surdéfini obligatoirement sous la forme d'une fonction amie, puisque son premier opérande n'est pas de type classe. L'opérateur de sortie dans un flot doit, lui aussi, être surdéfini sous la forme d'une fonction amie, mais pour une raison différente : son premier argument est de type `ostream`.

Voici la déclaration de notre classe `set_int` :

```
     /*  fichier SETINT.H  : déclaration de la classe set_int
*/
#include <iostream>
using namespace std ;
class set_int
{  int * adval ;                          // adresse du tableau
des valeurs
    int nmax ;                               // nombre maxi
d'éléments
    int nelem ;                           // nombre courant
```

```
d'éléments
 public :
   set_int (int = 20) ;               // constructeur
    set_int (set_int &) ;                 // constructeur par
recopie
                                        // voir remarque 1 ci-
après
      set_int  &  operator  =  (set_int  &)  ;  //  opérateur
d'affectation
                                        // voir remarque 2 ci-
après
   ~set_int () ;                           // destructeur
    int cardinal () ;                          // cardinal de
l'ensemble
    set_int & operator << (int) ;               // ajout d'un
élément
    friend int operator % (int, set_int &) ; // appartenance
d'un élément
                                             // voir remarque 3
ci-après
                 // envoi ensemble dans un flot, voir remarque
4
   friend ostream & operator << (ostream &, set_int &) ;
} ;
```

Voici ce que pourrait être la définition de notre classe (les points délicats sont commentés au sein même des instructions) :

```
#include "setint.h"
#include <iostream>
using namespace std ;
   /*************** constructeur ********************/
set_int::set_int (int dim)
{  adval = new int [nmax = dim] ;    // allocation tableau de
valeurs
   nelem = 0 ;
}
   /****************** destructeur ******************/
set_int::~set_int ()
{  delete [] adval ;                 // libération tableau de
valeurs
}
 /********** constructeur par recopie *************/
set_int::set_int (set_int & e)
{   adval = new int [nmax = e.nmax] ; // allocation nouveau
tableau
   nelem = e.nelem ;
```

```
   int i ;
    for (i=0 ; i<nelem ; i++)          // copie ancien tableau
dans nouveau
      adval[i] = e.adval[i] ;
}
   /************ opérateur d'affectation ************/
set_int & set_int::operator = (set_int & e)//commentaires fait
pour b = a
{  if (this != &e)                          // on ne fait rien
pour a = a
      { delete [] adval ;                        // libération
partie dynamique
                                              // de b
         adval = new int [nmax = e.nmax] ;    // allocation
nouvel ensemble
                                              // pour a
        nelem = e.nelem ;                     // dans lequel on
recopie
        int i ;                                  // entièrement
l'ensemble b
        for (i=0 ; i<nelem ; i++)             // avec sa partie
dynamique
          adval[i] = e.adval[i] ;
     }
   return * this ;
}
   /************ fonction membre cardinal ***********/
int set_int::cardinal ()
{  return nelem ;
}

   /************ opérateur d'ajout << ***************/
set_int & set_int::operator << (int nb)
{        // on examine si nb appartient déjà à l'ensemble
         // en utilisant l'opérateur %
          // s'il n'y appartient pas, et s'il y a encore de la
place
         // on l'ajoute à l'ensemble
    if ( ! (nb % *this) && nelem < nmax )  adval [nelem++] =
nb ;
   return (*this) ;
}

   /*********** opérateur d'appartenance % **********/
int operator % (int nb, set_int & e)
{  int i=0 ;
         // on examine si nb appartient déjà à l'ensemble
```

```
         //  (dans ce cas i vaudra nelem en fin de boucle)
   while ( (i<e.nelem) && (e.adval[i] != nb) ) i++ ;
   return (i<e.nelem) ;
}

   /****** opérateur << pour sortie sur un flot *****/
ostream & operator << (ostream & sortie, set_int & e)
{  sortie << "[ " ;
   int i ;
   for (i=0 ; i<e.nelem ; i++)
      sortie << e.adval[i] << " " ;
   sortie << "]" ;
   return sortie ;
}
```

Remarques

1. On pourrait ajouter le qualificatif `const` au constructeur par recopie, ce qui autoriserait l'initialisation d'un objet par un objet constant. Mais compte tenu des possibilités d'utilisation de l'autre constructeur dans des conversions implicites, on autoriserait du même coup l'initialisation par un entier ou un flottant, ce qui n'est guère satisfaisant ; on pourrait cependant interdire de telles possibilités en utilisant le mot-clé `explicit` dans la déclaration du constructeur.
2. La transmission de la valeur de retour de l'opérateur d'affectation n'est utile que si l'on souhaite permettre les affectations multiples. Il n'est pas indispensable de transmettre l'argument et la valeur de retour par référence, mais cela évite les recopies. On pourrait ici déclarer constant l'unique argument, ce qui autoriserait l'utilisation d'un objet constant en second opérande de l'affectation (moyennant une recopie). Mais, du même coup, compte tenu des possibilités de conversions implicites, on autoriserait également l'utilisation d'un entier ou d'un flottant, ce qui n'est pas nécessairement souhaité ; là encore, ces possibilités pourraient être interdites, moyennant l'utilisation appropriée du mot-clé `explicit`.
3. On pourrait ajouter le qualificatif `const` à l'opérateur `%`, ce qui permettrait de travailler sur un ensemble constant ; néanmoins, dans ce cas, il y aurait transmission d'une copie de cet ensemble à la fonction, ce qui n'est plus très efficace !

4. La remarque précédente s'applique à l'opérateur <<.
5. Notez qu'ici il n'est pas possible d'agrandir l'ensemble au-delà de la limite qui lui a été impartie lors de sa construction. Il serait assez facile de remédier à cette lacune en modifiant sensiblement la fonction d'ajout d'un élément (`operator <<`). Il suffirait, en effet, qu'elle prévoie, lorsque la limite est atteinte, d'allouer un nouvel emplacement dynamique, par exemple d'une taille double de l'emplacement existant, d'y recopier l'actuel contenu et de libérer l'ancien emplacement (en actualisant convenablement les membres donnée de l'objet).

Voici un exemple de programme utilisant la classe `set_int`, accompagné du résultat fourni par son exécution :

```
   /************* test de la classe set_int *********/
#include "setint.h"
#include <iostream>
using namespace std ;

main()
{  void fct (set_int) ;
   void fctref (set_int &) ;
   set_int ens ;
   cout << "donnez 10 entiers \n" ;
   int i, n ;
   for (i=0 ; i<10 ; i++)
      { cin >> n ;
        ens << n ;
      }
    cout << "il y a : " << ens.cardinal () << " entiers
différents\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fct (ens) ;
    cout << "au retour de fct, il y en a " << ens.cardinal ()
<< "\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fctref (ens) ;
    cout << "au retour de fctref, il y en a " << ens.cardinal
() << "\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   cout << "appartenance de -1  : " <<  -1 % ens << "\n" ;
   cout << "appartenance de 500 : " << 500 % ens << "\n" ;
   set_int ensa, ensb ;
   ensa = ensb = ens ;
```

```
   cout << "ensemble a : " << ensa << "\n" ;
   cout << "ensemble b : " << ensb << "\n" ;
}

void fct (set_int e)
{  cout << "ensemble reçu par fct : " << e << "\n" ;
   e << -1 << -2 << -3 ;
}

void fctref (set_int & e)
{  cout << "ensemble reçu par fctref : " << e << "\n" ;
   e << -1 << -2 << -3 ;
}
```

```
donnez 10 entiers
3 5 3 1 8 5 1 7 7 3
il y a : 5 entiers différents
qui forment l'ensemble : [ 3 5 1 8 7 ]
ensemble reçu par fct : [ 3 5 1 8 7 ]
au retour de fct, il y en a 5
qui forment l'ensemble : [ 3 5 1 8 7 ]
ensemble reçu par fctref : [ 3 5 1 8 7 ]
au retour de fctref, il y en a 8
qui forment l'ensemble : [ 3 5 1 8 7 -1 -2 -3 ]
appartenance de -1  : 1
appartenance de 500 : 0
ensemble a : [ 3 5 1 8 7 -1 -2 -3 ]
ensemble b : [ 3 5 1 8 7 -1 -2 -3 ]
```

Exercice 145

Énoncé

Créer une classe `vect` permettant de manipuler des « vecteurs dynamiques » d'entiers, c'est-à-dire des tableaux d'entiers dont la dimension peut être définie au moment de leur création (une telle classe a déjà été partiellement réalisée dans l'exercice 39). Cette classe devra disposer des opérateurs suivants :

- `[]` pour l'accès à une des composantes du vecteur, et cela aussi bien

au sein d'une expression qu'à gauche d'une affectation (mais cette dernière situation ne devra pas être autorisée sur des « vecteurs constants ») ;

- ==, tel que si `v1` et `v2` sont deux objets de type `vect`, `v1==v2` prenne la valeur `1` si `v1` et `v2` sont de même dimension et ont les mêmes composantes et la valeur `0` dans le cas contraire ;
- <<, tel que `flot<<v` envoie le vecteur `v` sur le flot indiqué, sous la forme :

```
<entier1, entier2, ... , entiern>
```

De plus, on s'arrangera pour que l'affectation et la transmission par valeur d'objets de type `vect` ne pose aucun problème ; pour ce faire, on acceptera de dupliquer complètement les objets concernés.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

Rappelons que lorsqu'on définit des objets constants, il n'est pas possible de leur appliquer une fonction membre publique, sauf si cette dernière a été déclarée avec le qualificatif `const` (auquel cas une telle fonction peut indifféremment être utilisée avec des objets constants ou non constants). Pour obtenir l'effet demandé de l'opérateur `[]`, lorsqu'il est appliqué à un vecteur constant, il est nécessaire d'en prévoir deux définitions dont l'une s'applique aux vecteurs constants ; pour éviter qu'on ne puisse, dans ce cas, l'utiliser à gauche d'une affectation, il est nécessaire qu'elle renvoie son résultat par valeur (et non par adresse comme le fera la fonction applicable aux vecteurs non constants).

Voici la déclaration de notre classe :

```
#include <iostream>
using namespace std ;
class vect
{  int nelem ;                        // nombre de composantes du
vecteur
   int * adr ;                             // pointeur sur partie
```

```
dynamique
 public :
   vect (int n=1) ;                // constructeur "usuel"
    vect (vect & v) ;                    // constructeur par
recopie,
                                   // voir remarque 1 ci-après
   ~vect () ;                      // destructeur
    friend ostream & operator << (ostream &, vect &) ; //
sortie sur un flot
   vect & operator = (vect & v) ;  // surdéfinition opérateur
affectation
                                   // voir remarque 2 ci-après
   int & operator [] (int i) ;     // surdef [] pour vect non
constants
    int   operator [] (int i) const ; // surdef [] pour vect
constants
} ;
```

Remarques

1. On pourrait ajouter le qualificatif `const` au constructeur par recopie, ce qui autoriserait l'initialisation d'un vecteur par un vecteur constant. Mais compte tenu des possibilités de l'autre constructeur dans des conversions implicites, on autoriserait du même coup l'initialisation par un entier ou un flottant, ce qui n'est guère satisfaisant ; on pourrait cependant interdire de telles possibilités en utilisant le mot-clé `explicit` dans la déclaration du constructeur.
2. La transmission de la valeur de retour de l'opérateur d'affectation n'est utile que si l'on souhaite permettre les affectations multiples. Il n'est pas indispensable de transmettre l'argument et la valeur de retour par référence, mais cela évite les recopies. On pourrait ici déclarer constant l'unique argument, ce qui autoriserait l'utilisation d'un objet constant en second opérande de l'affectation (moyennant une recopie). Mais, du même coup, compte tenu des possibilités de conversions implicites, on autoriserait également l'utilisation d'un entier ou d'un flottant, ce qui n'est pas nécessairement souhaité ; là encore, ces possibilités pourraient être interdites, moyennant l'utilisation appropriée du mot-clé `explicit`.

Voici la définition des différentes fonctions :

```
#include "vect.h"
#include <iostream>
using namespace std ;

vect::vect (int n)                    // constructeur "usuel"
     { adr = new int [nelem = n] ;
     }
vect::vect (vect & v)                      // constructeur par
recopie
     { adr = new int [nelem = v.nelem] ;
       int i ;
       for (i=0 ; i<nelem ; i++)
           adr[i] = v.adr[i] ;
     }
vect::~vect ()                         // destructeur
     { delete [] adr ;
     }
vect & vect::operator = (vect & v)     //  surdéfinition
opérateur affectation
     { if (this != &v)                  // on ne fait rien pour
a=a
         { delete [] adr ;
           adr = new int [nelem = v.nelem] ;
           int i ;
           for (i=0 ; i<nelem ; i++)
             adr[i] = v.adr[i] ;
         }
       return * this ;
     }
int & vect::operator [] (int i)          //  surdéfinition
opérateur []
     { return adr[i] ;
     }
int   vect::operator  []  (int  i)  const  //  surdéfinition
opérateur [] pour cst
     { return adr[i] ;
     }
ostream & operator << (ostream & sortie, vect & v)
     { sortie << "<" ;
       int i ;
       for (i=0 ; i<v.nelem ; i++) sortie << v.adr[i] << " " ;
       sortie << ">" ;
       return sortie ;
     }
```

À titre indicatif, voici un petit programme utilisant la classe `vect`, accompagné du résultat fourni par son exécution :

```
#include "vect.h"
#include <iostream>
using namespace std ;

main()
{  int i ;
   vect v1(5), v2(10) ;
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " << v1 << "\n" ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " << v2 << "\n" ;
   v1 = v2 ;
   cout << "v1 = " << v1 << "\n" ;
   vect v3 = v1 ;
   cout << "v3 = " << v3 << "\n" ;
   vect v4 = v2 ;
   cout << "v4 = " << v4 << "\n" ;
    // const vect w(3) ; w[2] = 5 ;  // conduit bien à erreur
compilation
}
```

```
 v1 = <0 1 2 3 4 >
 v2 = <0 1 4 9 16 25 36 49 64 81 >
 v1 = <0 1 4 9 16 25 36 49 64 81 >
 v3 = <0 1 4 9 16 25 36 49 64 81 >
 v4 = <0 1 4 9 16 25 36 49 64 81 >
```

Exercice 146

Énoncé

Réaliser une classe nommée `bit_array` permettant de manipuler des tableaux de bits (autrement dit, des tableaux dans lesquels chaque élément ne peut prendre que l'une des deux valeurs `0` ou `1`). La taille d'un tableau (c'est-à-dire le nombre de bits) sera définie lors de sa création (par un argument passé à son constructeur). On prévoira les opérateurs suivants :

- `+=`, tel que `t+=n` mette à `1` le bit de rang `n` du tableau `t` ;
- `-=`, tel que `t-=n` mette à `0` le bit de rang `n` du tableau `t` ;

- `[]`, tel que l'expression `t[i]` fournisse la valeur du bit de rang `i` du tableau `t` (on ne prévoira pas, ici, de pouvoir employer cet opérateur à gauche d'une affectation, comme dans `t[i] = ...`) ;
- `++`, tel `que t++` mette à `1` tous les bits de `t` ;
- `--`, tel que `t--`mette à `0` tous les bits de `t` ;
- `<<`, tel que `flot << t` envoie le contenu de *t* sur le flot indiqué, sous la forme :

```
<* bit1, bit2, ... bitn *>
```

On fera en sorte que l'affectation et la transmission par valeur d'objets du type `bit_array` ne pose aucun problème.

N.B. Le chapitre 21 vous montrera comment résoudre cet exercice à l'aide des composants standard introduits par la norme, qu'il ne faut pas chercher à utiliser ici.

Solution

Si l'on cherche à minimiser l'emplacement mémoire utilisé pour les objets de type `bit_array`, il est nécessaire de n'employer qu'un seul bit pour représenter un « élément » d'un tableau. Ces bits devront donc être regroupés, par exemple à raison de `CHAR_BIT` (défini dans `limits.h`) bits par caractère.

Manifestement, il faut prévoir que l'emplacement destiné à ces différents bits soit alloué dynamiquement en fonction de la valeur fournie au constructeur : pour `n` bits, il faudra `n/CHAR_BIT+1` caractères.

En membres donnée, il nous suffit de disposer d'un pointeur sur l'emplacement dynamique en question, ainsi que du nombre de bits du tableau. Pour simplifier certaines des fonctions membre, nous prévoirons également de conserver le nombre de caractères correspondant.

Les opérateurs `+=`, `-=`, `++` et `--` peuvent être définis indifféremment sous la forme d'une fonction membre ou d'une fonction amie. Ici, nous avons choisi des fonctions membre. L'énoncé ne précise rien quant au résultat fourni par ces 4 opérateurs. En fait, on pourrait prévoir qu'ils restituent le tableau après qu'ils y ont effectué l'opération voulue, mais en pratique, cela semble de peu

d'intérêt. Ici, nous avons donc simplement prévu que ces opérateurs ne fourniraient aucun résultat.

Pour que `[]` ne soit pas utilisable dans une affectation de la forme `t[i] = ...`, il suffit de prévoir qu'il fournisse son résultat par valeur (et non par référence comme on a généralement l'habitude de le faire).

Naturellement, ici encore, l'énoncé nous impose de surdéfinir l'opérateur d'affectation et de prévoir un constructeur par recopie.

Voici ce que pourrait être la déclaration de notre classe `bit_array` :

```
   /* fichier bitarray.h : déclaration de la classe bit_array
*/
#include <iostream>
using namespace std ;
class bit_array
{   int nbits ;       // nombre courant de bits du tableau
     int ncar ;          // nombre de caractères nécessaires
(redondant)
     char * adb ;       // adresse de l'emplacement contenant
les bits
  public :
    bit_array (int = 16) ;      // constructeur usuel
    bit_array (bit_array &) ;   // constructeur par recopie,
                                // voir remarque 1 ci-après
    ~bit_array () ;             // destructeur
                                // les opérateurs binaires
    bit_array & operator = (bit_array &) ;   // affectation,
                                              // voir remarque
2 ci-après
     int operator [] (int) ;                   // valeur d'un
bit
     void operator += (int) ;                   // activation
d'un bit
    void operator -= (int) ;                  // désactivation
d'un bit
                                             // envoi sur flot
    friend ostream & operator << (ostream &, bit_array &) ;

                                // les opérateurs unaires
    void operator ++ () ;                              // mise à
1
    void operator -- () ;                              // mise à
0
     void operator ~ () ;                                    //
complément à 1
```

```
} ;
```

Remarques

1. On pourrait ajouter le qualificatif `const` au constructeur par recopie, ce qui autoriserait l'initialisation d'un objet `bit_array` par un objet constant. Mais compte tenu des possibilités de conversions implicites, on autoriserait du même coup l'initialisation par un entier ou par un flottant, ce qui n'est guère satisfaisant ; on pourrait cependant interdire de telles possibilités en utilisant le mot-clé `explicit` dans le constructeur à un argument de type `int`.
2. La présence d'une valeur de retour dans l'opérateur d'affectation n'est utile que pour permettre les affectations multiples. La transmission par référence de l'unique argument n'est pas obligatoire. On pourrait ajouter le qualificatif `const` pour autoriser l'affectation d'un objet constant (moyennant alors une recopie). Mais, du même coup, compte tenu des possibilités de conversions implicites, on autoriserait également l'utilisation d'un entier ou d'un flottant, ce qui n'est pas nécessairement satisfaisant ; là encore, ces possibilités pourraient être interdites, moyennant l'utilisation appropriée du mot-clé `explicit`.

Voici la définition des différentes fonctions.

```
#include "bitarray.h"
#include <climits>
#include <iostream>
using namespace std ;

bit_array::bit_array (int nb)
{  nbits = nb ;
   ncar = nbits / CHAR_BIT + 1 ;
   adb = new  char [ncar] ;
   int i ;
   for (i=0 ; i<ncar ; i++) adb[i] = 0 ;    // raz
}
bit_array::bit_array (bit_array & t)
{  nbits = t.nbits ; ncar = t.ncar ;
   adb = new  char [ncar] ;
   int i ;
   for (i=0 ; i<ncar ; i++) adb[i] = t.adb[i] ;
```

```
}
bit_array::~bit_array()
{  delete [] adb ;
}

bit_array & bit_array::operator = (bit_array & t)
{  if (this != & t)      //  on ne fait rien pour t=t
     { delete [] adb ;
       nbits = t.nbits ; ncar = t.ncar ;
       adb = new char [ncar] ;
       int i ;
       for (i=0 ; i<ncar ; i++)
          adb[i] = t.adb[i] ;
     }
   return *this ;
}
int bit_array::operator [] (int i)
{       // le bit de rang i s'obtient en considérant le bit
          // de rang i % CHAR_BIT du caractère de rang i /
CHAR_BIT
    int carpos = i / CHAR_BIT ;
    int bitpos = i % CHAR_BIT ;
    return  ( adb [carpos] >> CHAR_BIT - bitpos -1 ) & 0x01 ;
}
void bit_array::operator += (int i)
{    int carpos = i / CHAR_BIT ;
       if (carpos < 0 || carpos >= ncar) return ;      //
protection
     int bitpos = i % CHAR_BIT ;
     adb [carpos] |= (1 << (CHAR_BIT - bitpos - 1) ) ;
}
void bit_array::operator -= (int i)
{    int carpos = i / CHAR_BIT ;
       if (carpos < 0 || carpos >= ncar) return ;      //
protection
     int bitpos = i % CHAR_BIT ;
     adb [carpos] &= ~(1 << CHAR_BIT - bitpos - 1) ;
}
ostream & operator << (ostream & sortie, bit_array & t)
{   sortie << "<* " ;
    int i ;
    for (i=0 ; i<t.nbits ; i++)
       sortie << t[i] << " " ;
    sortie << "*>" ;
    return sortie ;
}
void bit_array::operator ++ ()
```

```
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = 0xFFFF ;
}
void bit_array::operator -- ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = 0 ;
}

void bit_array::operator ~ ()
{   int i ;
    for (i=0 ; i<ncar ; i++) adb[i] = ~ adb[i] ;
}
```

Voici un programme d'essai de la classe `bit_array`, accompagné du résultat fourni par son exécution :

```
   /* programme d'essai de la classe bit_array */
main ()
{ bit_array t1 (34) ;
  cout << "t1 = " << t1 << "\n" ;
  t1 +=3 ; t1 += 0 ; t1 +=8 ; t1 += 15 ; t1 += 33 ;
  cout << "t1 = " << t1 << "\n" ;
  t1-- ;
  cout << "t1 = " << t1 << "\n" ;
  t1++ ;
  cout << "t1 = " << t1 << "\n" ;
  t1 -= 0 ; t1 -= 3 ; t1 -= 8 ; t1 -= 15 ; t1 -= 33 ;
  cout << "t1 = " << t1 << "\n" ;
  cout << "t1 = " << t1 << "\n" ;
  bit_array t2 (11), t3 (17) ;
  cout << "t2 = " << t2 << "\n" ;
  t2 = t3 = t1 ;
  cout << "t3 = " << t3 << "\n" ;
}
```

```
t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 *>
t1 = <* 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 *>
t1 = <* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 *>
t1 = <* 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 *>
t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 *>
t1 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 *>
t2 = <* 0 0 0 0 0 0 0 0 0 0 0 *>
```

```
t3 = <* 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 *>
```

Exercice 147

Énoncé

La capacité des nombres entiers est limitée par la taille du type `longint`. Créer une classe `big_int` permettant de manipuler des nombres entiers de **valeur absolument quelconque**.

Pour ne pas alourdir l'exercice, on se limitera à des nombres sans signe et à l'opération d'addition ; on s'arrangera toutefois pour que des expressions mixtes (c'est-à-dire mélangeant des objets de type `long_int` avec des entiers usuels) aient un sens.

On définira l'opérateur << pour qu'il permette d'envoyer un objet de type `big_int` sur un flot. Parmi les différents constructeurs, on en prévoira un avec un argument de type chaîne de caractères, correspondant aux chiffres d'un « grand entier ».

On fera en sorte que l'affectation et la transmission par valeur d'objets de type `big_int` ne posent aucun problème.

Solution

Pour représenter un « grand entier », la démarche la plus naturelle (mais pas la plus économique en place mémoire !) consiste à conserver le nombre sous forme décimale, à chaque chiffre étant associé un caractère. Pour ce faire, on peut choisir de « coder » un tel chiffre par le caractère correspondant (`'0'` pour `0`, `'1'` pour `1`…) ; on peut aussi choisir de placer une valeur égale au chiffre lui-même (`0` pour `0`, `1` pour `1`…). La dernière solution oblige à effecteur un « transcodage » lorsque l'on doit passer de la forme chaîne de caractères à la forme `big_int` (dans le constructeur correspondant, notamment) ou, inversement, lorsqu'on doit passer de la forme `big_int` à la forme suite de caractères (pour l'affichage). En revanche, elle simplifie

quelque peu l'algorithme d'addition, et c'est elle que nous avons choisie.

L'emplacement permettant de conserver un grand entier sera alloué dynamiquement ; sa taille sera, naturellement, adaptée à la valeur du nombre qui s'y trouvera. On conservera également le nombre courant de chiffres de l'entier ; on pourrait, en toute rigueur, s'en passer mais nous verrons que sa présence simplifie quelque peu la programmation. En ce qui concerne l'ordre de rangement des chiffres au sein de l'emplacement correspondant, il y a manifestement deux possibilités. Chacune possède des avantages et des inconvénients ; nous avons ici choisi de ranger les chiffres dans l'ordre inverse de celui où on les écrit (unités, dizaines, centaines...).

Pour pouvoir accepter les expressions mixtes, on dispose de plusieurs solutions :

- soit surdéfinir l'opérateur + pour tous les cas possibles ;
- soit surdéfinir + uniquement lorsqu'il porte sur des grands entiers et prévoir un constructeur recevant un argument de type `unsigned long` ; il permettra ainsi la conversion en `big_int` de n'importe quel type numérique.

C'est la seconde solution que nous avons adoptée. Notez toutefois que, si elle a le mérite d'être la plus simple à programmer, elle n'est pas la plus efficace en temps d'exécution.

Par ailleurs, pour que les conversions envisagées s'appliquent au premier opérande de l'addition, il est nécessaire de surdéfinir l'opérateur + comme une fonction amie.

Voici la déclaration de notre classe `big_int` (la présence d'un constructeur privé à deux arguments entiers sera justifiée un peu plus loin) :

```
    /* fichier bigint.h : déclaration de la classe big_int */
#define NCHIFMAX 32   // nombre maxi de chiffres d'un entier
(dépend de
                      // l'implémentation)
#include <iostream>
using namespace std ;

class big_int
{   int nchif ;             // nombre de chiffres
    char * adchif ;           // adresse emplacement contenant
les chiffres
     big_int (int, int) ;     // constructeur privé (à usage
```

```
interne)
  public :
     big_int (unsigned long=0) ;    // constructeur à partir
d'un nombre usuel
     big_int (char *) ;        // constructeur à partir d'une
chaîne
    big_int (big_int &) ;    // constructeur par recopie, voir
remarque 1
      big_int & operator = (const big_int &) ;          //
affectation,
                                                        // voir
remarque 2
     friend big_int operator + (const big_int &,const  big_int
&) ;
                                                             //
opérateur +,
                                                        // voir
remarque 3
     friend ostream & operator << (ostream &, big_int &) ; //
opérateur <<
} ;
```

Remarques

1. On pourrait ajouter le qualificatif `const` au constructeur par recopie, ce qui aurait pour effet d'autoriser l'initialisation d'un grand entier par un grand entier constant ou encore (compte tenu des possibilités de conversion implicite de l'opérande) par un entier, voire un flottant (par conversion en entier).
2. La transmission par référence du résultat de l'opérateur d'affectation autorise les affectations multiples. La transmission par référence de l'unique argument n'est pas obligatoire. On a ajouté le qualificatif `const` pour autoriser l'affectation d'une expression (notamment la somme de deux grands entiers) ; compte tenu des possibilités de conversions implicites, cela autorise du même coup l'affectation d'un entier, d'un flottant (par conversion en entier) ou d'un pointeur de type `char *` (encore faut-il, comme ici, ne pas avoir utilisé le mot-clé `explicit` dans les constructeurs correspondants).
3. Pour les arguments de l'opérateur `+`, le qualificatif `const` est indispensable si l'on souhaite bénéficier des conversions implicites des

opérandes, notamment d'un entier en un grand entier. Du coup, compte tenu des possibilités de conversions implicites des opérandes, on autorise également l'utilisation de flottants ou de pointeurs de type `char *`.

L'opérateur `+` commence par créer un emplacement temporaire pouvant recevoir un nombre comportant un chiffre de plus que le plus grand de ses deux opérandes (on ne sait pas encore combien de chiffres comportera exactement le résultat). On y calcule la somme suivant un algorithme calqué sur le processus manuel d'addition.

Puis on crée un objet de type `big_int` en utilisant un constructeur particulier : `big_int (int, int)`. En fait, nous avons besoin d'un constructeur créant un `big_int` comportant un nombre de chiffres donné, ce dont nous ne disposons pas dans les constructeurs publics. De plus, nous ne pouvons pas utiliser un constructeur de la forme `big_int (int)` car, alors, les additions mixtes faisant intervenir des entiers chercheraient à l'employer pour effectuer une conversion ! C'est pourquoi nous avons prévu un constructeur à deux arguments, le second étant fictif ; de plus, nous l'avons rendu privé, car il n'a nullement besoin d'être accessible à un utilisateur de la classe.

Voici la définition des fonctions de la classe `big_int` :

```
   /* définition des fonctions de la classe big_int */
#include <string.h>
#include "bigint.h"
#include <iostream>
using namespace std ;

big_int::big_int (int n, int p)     // l'argument p est fictif
{   nchif = n ;
    adchif = new char [nchif] ;
}

big_int::big_int (char * ch)
{
    nchif = strlen (ch) ;
    adchif = new char [nchif] ;
    int i ; char c ;
    for (i=0 ; i<nchif ; i++)
      { c = ch[i] - '0' ;
        if (c<0 || c>9) c=0 ;       // précaution
```

```
        adchif[nchif-i-1] = c ;     // attention à l'ordre des
chiffres !
      }
}

big_int::big_int (unsigned long n)
{      //  on  crée  le  grand  entier  correspondant  dans  un
emplacement temporaire
    char * adtemp = new char [NCHIFMAX] ;
    int i = 0 ;
    while (n)
      { adtemp [i++] = n % 10 ;
        n /= 10 ;
      }
       // ici i contient le nombre exact de chiffres
    nchif = i ;
    adchif = new char [nchif] ;
    for (i=0 ; i<nchif ; i++)
        adchif [i] = adtemp [i] ;
       // on libère l'emplacement temporaire
    delete [] adtemp ;
}

big_int::big_int (big_int & n)
{  nchif = n.nchif ;
   adchif = new char [nchif] ;
   int i ;
   for (i=0 ; i<nchif ; i++)
       adchif [i] = n.adchif [i] ;
}

big_int & big_int::operator = (const big_int & n)
{  if (this != &n)
      { delete [] adchif ;
        nchif = n.nchif ;
        adchif = new char [nchif] ;
        int i ;
        for (i=0 ; i<nchif ; i++)
           adchif [i] = n.adchif [i] ;
      }
   return * this ;
}

big_int operator + (const big_int & n, const big_int & p) //
voir remarque
{   int nchifmax = (n.nchif > p.nchif) ? n.nchif : p.nchif ;
    int ncar = nchifmax + 1 ;
```

```
          // préparation du résultat dans zone temporaire de
taille ncar
    char * adtemp = new char [ncar] ;
    int i, s, chif1, chif2 ;
    int ret = 0 ;
    for (i=0 ; i<nchifmax ; i++)
      { chif1 = (i<n.nchif) ? n.adchif [i] : 0 ;
        chif2 = (i<p.nchif) ? p.adchif [i] : 0 ;
        s = chif1 + chif2 + ret ;
        if (s>=10) { s -= 10 ;
                     ret = 1 ;
                   }
               else ret = 0 ;
        adtemp [i] = s ;
      }
    if (ret == 1) adtemp [ncar-1] = 1 ;
          else ncar-- ;
         // construction d'un objet de type big_int où l'on
recopie le résultat
    big_int res (ncar, 0) ;      // second argument fictif
    res.nchif = ncar ;
    for (i=0 ; i<ncar ; i++)
       res.adchif [i] = adtemp [i] ;
    delete [] adtemp ;
    return res ;
}

ostream & operator << (ostream & sortie, big_int & n)
{   int i ;
     for (i=n.nchif-1 ; i>=0 ; i--)         // attention à
l'ordre !
       sortie << (int)n.adchif [i] ;
    return sortie ;
}
```

Certains compilateurs imposent l'attribut `const` pour les arguments de `operator +`.

Voici un petit programme d'utilisation de la classe `big_int`, accompagné du résultat fourni par son exécution :

```
   /* programme d'essai */
#include "bigint.h"
```

```
#include <iostream>
using namespace std ;

main()
{  big_int n1(12) ; big_int n2(35) ; big_int n3 ;
   n3 = n1 + n2 ;
   cout << n1 << " + " << n2 << " = " << n3 << "\n" ;
                  big_int      n4      ("1234567890123456789"),
n5("9876543210987654321"), n6 ;
   n6 = n4 + n5 ;
   cout << n4 << " + " << n5 << " = " << n6 << "\n" ;
   cout << n6 << " + " << n1 << " = " << n6 + n1 << "\n" ;
    n2 = n4 + 5 ; // serait rejetée si operator = n'avait pas
argument const
   cout << n4 << "+5 = " << n2 << "\n" ;
   // ici une expression comme  n1 + "123" serait correcte et
de type big_int
   // ici une expression comme  n2 + 5.69  serait correcte et
de type big_int
}
```

```
 12 + 35 = 47
 1234567890123456789       +       9876543210987654321       =
 11111111101111111110
 11111111101111111110 + 12 = 11111111101111111122
 1234567890123456789+5 = 1234567890123456794
```

Exercice 148

Énoncé

Créer un patron de classes nommé `stack_gene`, permettant de manipuler des piles dont les éléments sont de type quelconque. Ces derniers seront conservés dans un emplacement alloué dynamiquement et dont la dimension sera fournie au constructeur (il ne s'agira donc pas d'un paramètre expression du patron). La classe devra comporter les opérateurs suivants :

- `<<`, tel que `p<<n` ajoute l'élément `n` à la pile `p` (si la pile est pleine, il ne se passera rien) ;

- `>>`, tel que `p>>n` place dans `n` la valeur du haut de la pile `p`, en la supprimant de la pile (si la pile est vide, il ne se passera rien) ;
- `++`, tel que `++p` vale `1` si la pile `p` est pleine et `0` dans le cas contraire ;
- `--`, tel que `--p` vale `1` si la pile `p` est vide et `0` dans le cas contraire ;
- `<<`, tel que, `flot` étant un flot de sortie, `flot << p` affiche le contenu de la pile `p` sur le flot sous la forme : `// valeur_1 valeur_2... valeur_n //`.

On supposera que les objets de type `stack_gene` ne seront jamais soumis à des transmissions par valeur ou à des affectations ; on ne cherchera donc pas à surdéfinir le constructeur par recopie ou l'opérateur d'affectation.

En fait, on peut s'inspirer de ce qui a été fait dans l'exercice 95 pour réaliser une pile d'entiers, en faisant en sorte que `int` soit remplacé par un paramètre de type.

Voici ce que pourrait être la définition de notre patron de classes :

```
#include <stdlib.h>
#include <iostream>   // voir N.B. du paragraphe Nouvelles
possibilités
                      // d'entrées-sorties du chapitre 2
using namespace std ;

template <class T> class stack_gene
{   int nmax ;                 // nombre maximum de la valeur
de la pile
    int nelem ;               // nombre courant de valeurs de
la pile
    T * adv ;                // pointeur sur les valeurs
  public :
    stack_gene (int = 20) ;           // constructeur
    ~stack_gene () ;                  // destructeur
    stack_gene & operator << (T) ;    // opérateur d'empilage
    stack_gene & operator >> (T &) ;  // opérateur de dépilage
                                      // (attention T &)
     int operator ++ () ;                 // opérateur de test
pile pleine
     int operator -- () ;                 // opérateur de test
pile vide
```

```
                                                  // opérateur << pour
flot de sortie
     friend ostream & operator << (ostream &, stack_gene<T>
&) ;
} ;

template <class T> stack_gene<T>::stack_gene (int n)
{   nmax = n ;
    adv = new T [nmax] ;
    nelem = 0 ;
}
template <class T> stack_gene<T>::~stack_gene ()
{   delete [] adv ;
}
template <class T> stack_gene<T> & stack_gene<T>::operator <<
(T n)
{   if (nelem < nmax) adv[nelem++] = n ;
    return (*this) ;
}
template <class T> stack_gene<T> & stack_gene<T>::operator >>
(T & n)
{   if (nelem > 0) n = adv[--nelem] ;
    return (*this) ;
}
template <class T> int stack_gene<T>::operator ++ ()
{   return  (nelem == nmax) ;
}
template <class T> int stack_gene<T>::operator -- ()
{   return (nelem == 0) ;
}
template <class T> ostream & operator << (ostream & sortie,
stack_gene<T> & p)
{ sortie << "// " ;
  int i ;
  for (i=0 ; i<p.nelem ; i++) sortie << p.adv[i] << " " ;
  sortie << "//" ;
  return sortie ;
}
```

À titre indicatif, voici un petit programme d'utilisation de notre patron de classes (dont on suppose que la définition figure dans `stackg.h`). Il est accompagné du résultat fourni par son exécution.

```
   /************ programme d'essai de stack_gene *********/
#include "stackg.h"
#include <iostream>   // voir N.B. du paragraphe Nouvelles
possibilités
```

```
                    // d'entrées-sorties du chapitre 2
using namespace std ;
main()
{   stack_gene <int> pi(20) ;           // pile de 20 entiers
maxi
     cout << "pi pleine : " << ++pi << " vide : " << --pi <<
"\n" ;
    pi << 2 << 3 << 12 ;
    cout << "pi = " << pi << "\n" ;
      stack_gene <float> pf(10) ;            // pile de 10
flottants maxi
    pf << 3.5 << 4.25 << 2 ;            // 2 sera converti en
float
    cout << "pf = " << pf << "\n" ;
    float x ;  pf >> x ;
    cout << "haut de la pile pf = " << x ;
    cout << "pf = " << pf << "\n" ;
}
```

```
 pi pleine : 0 vide : 1
 pi = // 2 3 12 //
 pf = // 3.5 4.25 2 //
 haut de la pile pf = 2pf = // 3.5 4.25 //
```

Exercice 149

Énoncé

En s'inspirant de l'exercice 145, créer un patron de classes permettant de manipuler des vecteurs dynamiques dont les éléments sont de type quelconque.

Solution

Voici la déclaration de notre patron :

```
#include <iostream>
using namespace std ;
```

```
template <class T> class vect
{  int nelem ;                    // nombre de composantes du
vecteur
    T * adr ;                          // pointeur sur partie
dynamique
 public :
   vect (int n=1) ;              // constructeur "usuel"
   vect (vect & v) ;             // constructeur par recopie
    ~vect () ;                   // destructeur
   template <class U> friend ostream & operator << (ostream &,
                                                    vect <U>
&) ;
    vect & operator = (vect & v) ;     // surdéf opérateur
d'affectation
   T & operator [] (int i) ;          // surdéf [] pour vect
non constants
   T   operator [] (int i) const ;    // surdéf [] pour vect
constants
} ;
```

Voici la définition des différentes fonctions membre :

```
#include "vectgen.h"
#include <iostream>
using namespace std ;

template <class T> vect<T>::vect (int n)      // constructeur
"usuel"
     { adr = new T [nelem = n] ;
     }
template  <class  T>  vect<T>::vect  (vect<T>  &  v)    //
constructeur par recopie
     { adr = new T [nelem = v.nelem] ;
       int i ;
       for (i=0 ; i<nelem ; i++)
           adr[i] = v.adr[i] ;
     }
template <class T> vect<T>::~vect ()
     { delete [] adr ;
     }
template <class T> vect<T> & vect<T>::operator = (vect<T> & v)
     { if (this != &v)                          // on ne fait
rien pour a=a
         { delete [] adr ;
           adr = new T [nelem = v.nelem] ;
           int i ;
           for (i=0 ; i<nelem ; i++)
             adr[i] = v.adr[i] ;
```

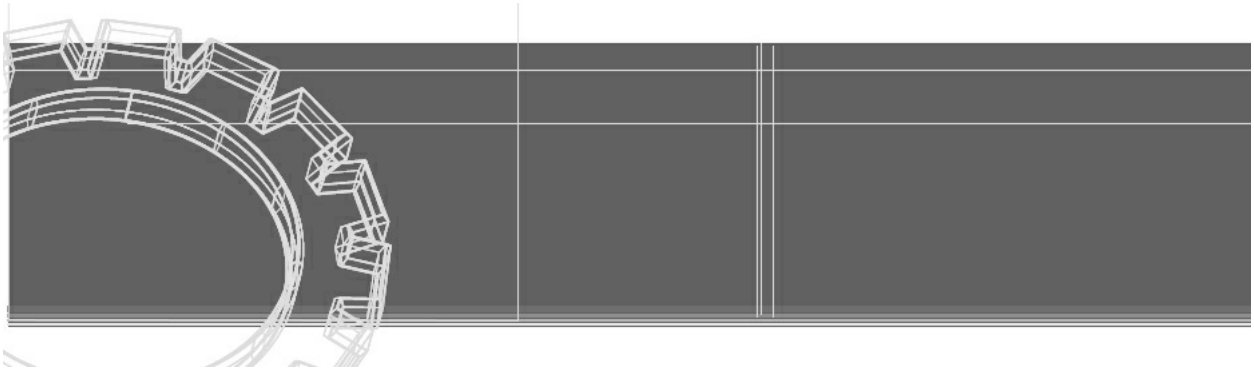
```
        }
      return * this ;
    }
template <class T> T & vect<T>::operator [] (int i)
    { return adr[i] ;
    }
template <class T> T   vect<T>::operator [] (int i) const
    { return adr[i] ;
    }
template <class T> ostream & operator << (ostream & sortie,
vect<T> & v)
    { sortie << "<" ;
      int i ;
      for (i=0 ; i<v.nelem ; i++) sortie << v.adr[i] << " " ;
      sortie << ">" ;
      return sortie ;
    }
```

À titre indicatif, voici un petit programme utilisant notre patron, accompagné du résultat fourni par son exécution :

```
#include "vectgen.h"
#include <iostream>
using namespace std ;
main()
{  int i ;
   vect <int> v1(5) ; vect <int> v2(10) ;
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " << v1 << "\n" ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " << v2 << "\n" ;
   v1 = v2 ;
   cout << "v1 = " << v1 << "\n" ;
   vect <int> v3  = v1 ;
// vect <double> v3 = v1 ;              // serait rejeté
   cout << "v3 = " << v3 << "\n" ;
// const vect <float> w(3) ; w[2] = 5 ; // conduit bien à
erreur compilation
// vect <float> v4(5) ; v4 = v1 ;         // conduit bien à
erreur compilation
}
```

```
 v1 = <0 1 2 3 4 >
 v2 = <0 1 4 9 16 25 36 49 64 81 >
 v1 = <0 1 4 9 16 25 36 49 64 81 >
 v3 = <0 1 4 9 16 25 36 49 64 81 >
```

Chapitre 21
Les composants standard



Les exercices des précédents chapitres ont été volontairement résolus sans recourir aux composants standard introduits par la norme de C++. Manifestement, l'existence de ces composants influe sur certains des exercices, soit en rendant leur solution quasiment triviale, soit en la simplifiant notablement. C'est ce que nous proposons d'examiner ici. Pour faciliter les choses, nous avons systématiquement reproduit le texte intégral de l'énoncé correspondant, avec son ancienne numérotation.

Notez que, contrairement aux autres chapitres, celui-ci n'a pas été doté d'un résumé. D'une part, l'ampleur de la bibliothèque de composants standard l'aurait rendu relativement volumineux (on en trouvera une étude détaillée dans l'un de nos ouvrages consacrés au langage C++ et publiés également aux éditions Eyrolles). D'autre part, il ne constitue pas à proprement parler un ensemble complet d'exercices sur le sujet.

Exercice 150 (69 revisité)

Ancien énoncé

Réaliser une classe nommée `set_char` permettant de manipuler des ensembles de caractères. On devra pouvoir réaliser sur un tel ensemble les opérations classiques suivantes : lui ajouter un nouvel élément, connaître son « cardinal » (nombre d'éléments), savoir si un caractère donné lui appartient.

Ici, on n'effectuera aucune allocation dynamique d'emplacements mémoire. Il faudra donc prévoir, en membre donnée, un tableau de taille fixe.

Écrire, en outre, un programme (`main`) utilisant la classe `set_char` pour déterminer le nombre de caractères différents contenus dans un mot lu en donnée.

Commentaires

Le conteneur `set<char>` peut jouer le rôle de la classe demandée `set_char`, à condition de supprimer de l'énoncé la contrainte relative à l'absence d'allocation dynamique. En effet, elle n'a plus de raison d'être, les fonctions membre de la classe `set<char>` allouant automatiquement la place nécessaire au fur et à mesure des besoins.

Voici ce que pourrait devenir le programme de test fourni précédemment dans le chapitre 3. On notera que la fonction membre `size` fournit le nombre d'éléments de l'ensemble, tandis que la fonction membre `insert` permet tout naturellement l'insertion d'un élément. Quant à la fonction `count`, elle fournit le nombre d'éléments de valeur donnée figurant dans l'ensemble ; son résultat est donc toujours soit `0`, soit `1`.

```
#include <iostream>
#include <set>          // pour la classe set
using namespace std ;
main()
{  set<char> ens ;     // voir remarque 1 ci-après
   char mot [81] ;
   cout << "donnez un mot : " ;
   cin >> mot ;
   int i ;
   for (i=0 ; i<strlen(mot) ; i++) ens.insert(mot[i]) ;
```

```
    cout << "il contient " << ens.size() << " caractères
différents\n" ;
      if (ens.count('e')) cout << "le caractère e est
présent\n" ;
                    else cout << "le caractère e n'est pas
présent\n" ;
}
```

```
donnez un mot : bonjour
il contient 6 caractères différents
le caractère e n'est pas présent
```

Remarques

1. Certains compilateurs imposent la présence d'un second paramètre de type précisant la relation d'ordre utilisée pour ordonner l'ensemble. Dans ce cas, il faudra écrire : `set<char, less<char> > ens`.
2. Il existe un autre conteneur, `multiset`, semblable à `set`, dans lequel une même valeur peut apparaître plusieurs fois. Il ne correspond plus à la notion mathématique d'ensemble. On notera que la fonction membre `count`, présente également dans ce conteneur, voit alors son nom nettement plus justifié que dans le cas de `set`.

Exercice 151 (70 revisité)

Ancien énoncé

Modifier la classe `set_char` précédente, de manière à disposer de ce que l'on nomme un « itérateur » sur les différents éléments de l'ensemble. Il s'agit d'un mécanisme permettant d'accéder séquentiellement aux différents éléments. On prévoira trois nouvelles fonctions membre : `init`, qui initialise le processus d'exploration ; `prochain`, qui fournit l'élément suivant lorsqu'il existe et `existe`, qui précise s'il existe encore un élément non exploré.

On complétera alors le programme d'utilisation précédent, de manière qu'il affiche les différents caractères contenus dans le mot fourni en donnée.

Commentaires

Ici encore, on peut utiliser le composant standard `set<char>` qui dispose d'un itérateur intégré `set<char>::iterator`. Bien entendu, il n'y a plus de raison d'imposer l'existence des fonctions `init`, `prochain` et `existe`. Les fonctions membre `begin` et `end` fournissent les valeurs initiales et finales à utiliser pour explorer l'ensemble à l'aide d'un tel itérateur. On prendra garde au fait que `end` pointe non pas sur le dernier élément du conteneur, mais juste après. L'avancement de l'itérateur s'obtient par l'opérateur `++`.

Voici ce que pourrait devenir le programme de test fourni précédemment. La notation `*ie` correspond à l'élément désigné par la valeur courante de l'itérateur `ie`.

```
#include <iostream>
#include <string.h>
#include <set>
using namespace std ;

main()
{  set<char> ens ;   // voir remarque 1 ci-après
   char mot [81] ;
   cout << "donnez un mot : " ;
   cin >> mot ;
   int i ;
   for (i=0 ; i<strlen(mot) ; i++) ens.insert(mot[i]) ;
     cout << "il contient " << ens.size() << " caractères
différents"
        << " qui sont :\n" ;
   set<char>::iterator ie ;    // itérateur sur un ensemble de
caractères
   for (ie=ens.begin() ; ie != ens.end() ; ie++)
      cout << *ie << " " ;
}
```

```
donnez un mot : bonjour
il contient 6 caractères différents qui sont :
b j n o r u
```

Remarques

1. Certains compilateurs imposent la présence d'un second paramètre de type précisant la relation d'ordre utilisée pour ordonner l'ensemble. Dans ce cas, il faudra écrire : `set<char, less<char> > ens`.
2. Il existe plusieurs sortes d'itérateurs. Les plus courants sont les itérateurs bidirectionnels qui peuvent être incrémentés (`++`) ou décrémentés (`--`) d'une position à la fois, et les itérateurs à accès direct qui permettent l'accès direct à un élément quelconque. Tous les conteneurs disposent des fonctions membre `begin` et `end` permettant leur parcours par un itérateur bidirectionnel.

Exercice 152 (79 revisité)

Ancien énoncé

1. Réaliser une classe nommée `set_int` permettant de manipuler des ensembles de nombres entiers. On devra pouvoir réaliser sur un tel ensemble les opérations classiques suivantes : lui ajouter un nouvel élément, connaître son cardinal (nombre d'éléments), savoir si un entier donné lui appartient.

   Ici, on conservera les différents éléments de l'ensemble dans un tableau alloué dynamiquement par le constructeur. Un argument (auquel on pourra prévoir une valeur par défaut) lui précisera le nombre maximal d'éléments de l'ensemble.
2. Écrire, en outre, un programme (`main`) utilisant la classe `set_int` pour déterminer le nombre d'entiers différents contenus dans un tableau d'entiers lus en données.
3. Que faudrait-il faire pour qu'un objet du type `set_int` puisse être transmis par valeur, soit comme argument d'appel, soit comme valeur de retour d'une fonction ?

Commentaires

On peut utiliser le composant `set<int>`, à condition de ne plus préciser qu'on conserve les éléments dans un tableau dynamique. La question 3 n'a plus de raison d'être, car la classe `set` dispose d'un constructeur par recopie.

Voici ce que devient l'exemple de programme demandé dans la seconde question :

```
#include <iostream>
#include <set>        // pour la classe set
using namespace std ;
main()
{  set<int> ens ;    // voir remarque ci-après
   cout << "donnez 20 entiers \n" ;
   int i, n ;
   for (i=0 ; i<20 ; i++)
   { cin >> n ;
     ens.insert (n) ;
   }
      cout << "il y a : " << ens.size() << " entiers
différents\n" ;
}
```

Remarque

Certains compilateurs imposent la présence d'un second paramètre de type précisant la relation d'ordre utilisée pour ordonner l'ensemble. Dans ce cas, il faudra écrire : `set<char, less<char> > ens`.

Exercice 153 (80 revisité)

N.B. Cet exercice n'est rappelé ici que parce qu'il est utile à l'énoncé suivant.

Ancien énoncé

Modifier l'implémentation de la classe précédente (avec son constructeur par recopie) de façon que l'ensemble d'entiers soit maintenant représenté

par une **liste chaînée** (chaque entier est rangé dans une structure comportant un champ destiné à contenir un nombre et un champ destiné à contenir un pointeur sur la structure suivante). L'interface de la classe (la partie publique de sa déclaration) devra rester inchangée, ce qui signifie qu'un client de la classe continuera à l'employer de la même façon.

Commentaires

Il s'agit ici d'une demande de modification d'implémentation d'une classe qui n'a manifestement aucun sens dans le cas d'un composant standard.

Exercice 154 (81 revisité)

Ancien énoncé

Modifier la classe `set_int` précédente (implémentée sous la forme d'une liste chaînée, avec ou sans son constructeur par recopie) pour qu'elle dispose de ce que l'on nomme un « itérateur » sur les différents éléments de l'ensemble. Rappelons qu'il s'agit d'un mécanisme permettant d'accéder séquentiellement aux différents éléments de l'ensemble. On prévoira trois nouvelles fonctions membre : `init`, pour initialiser le processus d'itération ; `prochain`, pour fournir l'élément suivant lorsqu'il existe et `existe`, pour tester s'il existe encore un élément non exploré.

On complétera alors le programme d'utilisation précédent (en fait, celui de l'exercice 26), de manière qu'il affiche les différents entiers contenus dans les valeurs fournies en donnée.

Commentaires

Ici encore, il n'y a aucune raison de vouloir modifier l'implémentation d'un composant standard. Voici ce que pourrait devenir l'exemple de programme d'utilisation si l'on utilisait le composant `set<int>` et l'itérateur associé `set<int>::iterator` :

```
#include <iostream>
#include <set>       // pour la classe set
using namespace std ;
main()
{  set<int> ens ;
   cout << "donnez 20 entiers \n" ;
   int i, n ;
   for (i=0 ; i<20 ; i++)
   { cin >> n ;
      ens.insert (n) ;
   }
      cout  <<  "il  y  a  :  "  <<  ens.size()  <<  "  entiers
différents\n" ;
   cout << "Ce sont : \n" ;
   set<int>::iterator is ;
   for (is=ens.begin() ; is != ens.end() ; is++)
      cout << *is << " " ;
}
```

Exercice **155** (92 revisité)

Ancien énoncé

Définir une classe `vect` permettant de représenter des « vecteurs dynamiques », c'est-à-dire dont la dimension peut ne pas être connue lors de la compilation. Plus précisément, on prévoira de déclarer de tels vecteurs par une instruction de la forme :

```
vect t(exp) ;
```

dans laquelle `exp` désigne une expression quelconque (de type entier).

On définira, de façon appropriée, l'opérateur `[]` de manière qu'il permette d'accéder à des éléments d'un objet d'un type `vect` comme on le ferait avec un tableau classique.

On ne cherchera pas à résoudre les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets de type `vect`. En revanche, on s'arrangera pour qu'aucun risque de « débordement » d'indice n'existe.

Commentaires

La classe `vector<int>` fait l'affaire, y compris pour l'affectation ou la transmission par valeur. Elle dispose d'un itérateur à accès direct (nommé toujours `iterator`). Mais, de plus, l'opérateur `[]` y est surdéfini de sorte qu'il fournit une écriture plus concise pour l'accès direct à un élément. Si `v` est un objet de type `vector<int>`, la notation `v[i]` est équivalente à `*(v.begin()+i)`.

Toutefois, cet opérateur `[]` n'est pas protégé contre les débordements d'indice. On peut résoudre le problème en utilisant, à sa place, la fonction membre `at` qui déclenche une exception standard `out_of_range` en cas de débordement d'indice. Si l'on veut absolument se tenir à l'interface imposée par l'énoncé, on peut également créer artificiellement une classe `vect` dérivée de `vector<int>`, dans laquelle on définit l'opérateur `[]` de façon appropriée. Cette dernière démarche a le mérite d'offrir toute latitude quant au traitement à mettre en œuvre en cas de débordement : déclenchement d'une exception, modification autoritaire de la valeur de l'indice comme on l'a fait dans la solution de l'exercice 39, etc.

Voici un premier exemple où l'on se contente d'utiliser la classe `vector<int>` et son opérateur `[]` :

```
#include <iostream>
#include <vector>        // pour la classe vector
using namespace std ;
main()
{  vector <int> v(6) ;
   int i ;
   for (i=0 ; i<6 ; i++) v[i] = i ;
   for (i=0 ; i<6 ; i++) cout << v[i] << " " ;
}
```

Voici un second exemple (accompagné du résultat fourni par son exécution) qui montre comment utiliser la fonction `at` de la classe `vector<int>` et traitant de façon appropriée l'exception `standard out_of_range` :

```
#include <iostream>
#include <vector>        // pour la classe vector
#include  <stdexcept>          //  pour  la  classe  exception
out_of_range
using namespace std ;
main()
{ try
```

```
  { vector <int> v(6) ;
     int i ;
     for (i=0 ; i<6 ; i++) v.at(i) = i ;
      for (i=0 ; i<8 ; i++) cout << v.at(i) << " " ; // ici on
déborde de v
  }
  catch (out_of_range oor)
  { cout << "exception out of range\n" ;
    exit(-1) ;
  }
}
```

```
 0 1 2 3 4 5 exception out of range
```

Voici un troisième exemple dans lequel on crée une classe `vect` dérivée de `vector<int>`. Ici, on déclenche une exception `out_of_range` en cas d'indice incorrect. Notez qu'il est nécessaire de redéfinir le constructeur à un argument entier de `vect`, bien que son corps soit vide, ceci afin de transmettre la dimension au constructeur de la classe de base. Il faudrait d'ailleurs faire de même pour tout constructeur de `vect` avec arguments qu'on souhaiterait pouvoir utiliser.

```
#include <iostream>   // voir N.B. du paragraphe Nouvelles
possibilités
                      // d'entrées-sorties du chapitre 2
#include <vector>     // pour la classe vector
#include <stdexcept> // pour la classe exception out_of_range
using namespace std ;

class vect : public vector<int>
{ public :
   vect (int dim) : vector<int> (dim) {}  // indispensable
   // surdéfinition de l'opérateur []
   int & operator [] (int i)
    { // on pourrait aussi chercher à modifier autoritairement
la valeur
     // de i par :
     // if ( (i<0)  || (i>=(*this).size()) ) i=0 ;
     return (*this).at(i) ;
   }
} ;

main()
{ try
  { vect v(6) ;
```

```
      int i ;
      for (i=0 ; i<6 ; i++) v[i] = i ;
       for (i=0 ; i<8 ; i++) cout << v[i] << " " ; // ici on
déborde de v
  }
  catch (out_of_range oor)
  { cout << "exception out of range\n" ;
    exit(-1) ;
  }
}
```

```
  0 1 2 3 4 5 exception out of range
```

Exercice 156 (93 revisité)

Ancien énoncé

En s'inspirant de l'exercice précédent, on souhaite créer une classe `int2d` permettant de représenter des tableaux dynamiques d'entiers à deux indices, c'est-à-dire dont les dimensions peuvent ne pas être connues lors de la compilation. Plus précisément, on prévoira de déclarer de tels tableaux par une déclaration de la forme :

```
int2d t(exp1, exp2) ;
```

dans laquelle `exp1` et `exp2` désignent une expression quelconque (de type entier).

On surdéfinira l'opérateur `()`, de manière qu'il permette d'accéder à des éléments d'un objet d'un type `int2d` comme on le ferait avec un tableau classique.

Là encore, on ne cherchera pas à résoudre les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets de type `int2d`. En revanche, on s'arrangera pour qu'il n'existe aucun risque de débordement d'indice.

Commentaires

On peut facilement généraliser ce qui a été fait dans l'exercice précédent. La classe `vector<vector<int> >` fait l'affaire, y compris pour l'affectation ou la transmission par valeur. Mais, là encore, l'opérateur `[]` n'est pas protégé contre les débordements d'indice. On peut résoudre le problème en utilisant, à sa place, la fonction membre `at` qui déclenche une exception standard `out_of_range` en cas de débordement d'indice.

Dans le programme source, il faudra absolument laisser un espace entre les deux symboles >, afin d'éviter toute confusion avec l'opérateur >>.

Voici un premier exemple (accompagné du résultat fourni par son exécution) où l'on se contente d'utiliser la classe `vector<vector<int> >` et son opérateur `[]` :

```
#include <iostream>
#include <vector>
using namespace std ;

main()
{ vector<vector<int> > t1 (4) ;  // vecteur de 4 vecteurs
  vector<int> v(3) ;              // vecteur de 3 entiers, non
initialisé
  int i, j ;
  for (i=0 ; i<4 ; i++)
    t1[i] = v ;
  for (i=0 ; i<4 ; i++)
    for (j=0 ; j<3 ; j++)
    t1[i][j] = i+j ;
  for (i=0 ; i<4 ; i++)
    { for (j=0 ; j<3 ; j++)
    cout << t1 [i] [j] << " " ;
    cout << "\n" ;
    }
}
```

```
 0 1 2
 1 2 3
```

```
2 3 4
3 4 5
```

On notera qu'on a quand même dû créer un objet temporaire `v`, de type `vector<int>`, afin d'initialiser les différents vecteurs de `t1`. Mais les choses restent néanmoins plus simples que ce qui a été fait au chapitre 7, sans recourir aux composants standard.

Voici un second exemple, qui montre comment utiliser la fonction `at` des classes `vector<int>` et `vector <vector<int>>` et traitant de façon appropriée l'exception `out_of_range` :

```
#include <iostream>
#include <vector>
using namespace std ;
main()
{ try
   { vector<vector<int> > t1 (4) ; // vecteur de 4 vecteurs -
espace
                                   // entre > et >
    vector<int> v(3) ;             // vecteur de 3 entiers, non
initialisé
    int i, j ;

    for (i=0 ; i<4 ; i++)
      t1.at(i) = v ;
    for (i=0 ; i<4 ; i++)
      for (j=0 ; j<3 ; j++)
        (t1.at(i)).at(j) = i+j ;
    for (i=0 ; i<4 ; i++)
      { for (j=0 ; j<3 ; j++)
            cout << (t1.at(i)).at(j) << " " ;
        cout << "\n" ;
      }
  }
  catch (out_of_range oor)
    { cout << "exception out of range\n" ;
      exit(-1) ;
    }
}
```

On peut aussi, à l'image de ce que l'on a fait dans l'exercice précédent, chercher à se tenir à l'interface imposée par l'énoncé. Dans ce cas, on crée artificiellement une classe `int2d`, dérivée de `vector<vector<int>`, classe de base dont on exploite les fonctionnalités comme le montre l'exemple suivant.

Ici, nous avons attribué des valeurs arbitraires aux indices en cas de débordement, comme au chapitre 7.

```
#include <iostream>
#include <vector>
using namespace std ;

   /******** déclaration de la classe int2d ********/
class int2d : public vector<vector<int> >
{  int nlig ;        // nombre de "lignes"
   int ncol ;        // nombre de "colonnes"
  public :
   int2d (int nl, int nc) ;    // constructeur
    int & operator () (int, int) ; // accès à un élément, par
ses 2 "indices"
} ;

   /*********** définition du constructeur **********/
int2d::int2d (int nl, int nc) : vector<vector<int> > (nl)
{  nlig = nl ; ncol = nc ;
   vector<int> v(nc) ;
   int i ;
   for (i=0 ; i<nl ; i++) (*this)[i] = v ;
}

   /********** définition de l'opérateur () *********/
int & int2d::operator () (int i, int j)
{  if ( (i<0) || (i>=nlig) ) i=0 ;  // protections sur premier
indice
   if ( (j<0) || (j>=ncol) ) j=0 ;  // protections sur second
indice
   return (*this)[i][j] ;
}

main()
{   int2d t1 (4,3) ;
    int i, j ;
    for (i=0 ; i<4 ; i++)
       for (j=0 ; j<3 ; j++)
    t1(i, j) = i+j ;
    for (i=0 ; i<4 ; i++)
       { for (j=0 ; j<3 ; j++)
              cout << t1 (i, j) << " " ;
         cout << "\n" ;
       }
}
```

La comparaison entre cette solution et celle du chapitre 7 montre que le gain

obtenu avec l'utilisation des composants standard reste assez relatif. Toutefois, il faut voir que dans une situation réelle, la solution du chapitre 7 nécessiterait la redéfinition de l'affectation et du constructeur par recopie de `int2d`. Cela ne serait pas nécessaire ici, les fonctions correspondantes de la classe de base faisant l'affaire puisque la classe dérivée ne comporte aucune partie dynamique supplémentaire.

Exercice 157 (95 revisité)

Ancien énoncé

Réaliser une classe nommée `stack_int` permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un emplacement alloué dynamiquement ; sa dimension sera déterminée par l'argument fourni à son constructeur (on lui prévoira une valeur par défaut de 20). Cette classe devra comporter les opérateurs suivants (nous supposons que `p` est un objet de type `stack_int` et `n` un entier) :

- `<<`, tel que `p<<n` ajoute l'entier `n` à la pile `p` (si la pile est pleine, rien ne se passe) ;
- `>>`, tel que `p>>n` place dans `n` la valeur du haut de la pile `p`, en la supprimant de la pile (si la pile est vide, la valeur de `n` ne sera pas modifiée) ;
- `++`, tel que `p++` vale `1` si la pile `p` est pleine et `0` dans le cas contraire ;
- `--`, tel que `p--` vale `1` si la pile `p` est vide et `0` dans le cas contraire.

On prévoira que les opérateurs `<<` et `>>` pourront être utilisés sous les formes suivantes (`n1`, `n2` et `n3` étant des entiers) :

```
p << n1 << n2 << n3 ;          p >> n1 >> n2 << n3 ;
```

On fera en sorte qu'il soit possible de transmettre une pile par valeur. En revanche, l'affectation entre piles ne sera pas permise, et on s'arrangera pour que cette situation aboutisse à un arrêt de l'exécution.

Commentaires

Si l'on ne s'intéresse qu'aux seules fonctionnalités (ajout, extraction, suppression, test pile pleine ou pile vide) de la classe qu'on demande de créer, il existe un composant qui fait l'affaire. Il s'agit de `stack<int, vector<int>>`. Ici, on s'attendrait plus simplement à `stack<int>`. En fait, le patron `stack` est non pas un conteneur à part entière, mais ce que l'on appelle un adaptateur de conteneur. Il s'agit d'un patron de classes, fondé sur un conteneur d'un type donné (ici `vector<int>`) qui en modifie l'interface, à la fois en la restreignant et en l'adaptant à des fonctionnalités données, à savoir ici :

- test pile vide (fonction `empty`) ;
- accès à l'information située au somment de la pile (fonction `top`) : cette fonction ne modifie pas la valeur du sommet ;
- dépôt d'une valeur sur la pile (fonction `push`) ;
- suppression de la valeur située au sommet de la pile (fonction `pop`) ; on notera que pour véritablement « dépiler » une valeur, il faut effectuer deux appels : `top` puis `pop`.

Cet adaptateur, `vector<int>`, peut se baser sur `vector`, `deque` ou `list`. Nous ne justifierons pas ici le choix de `vector`, dicté uniquement par des détails d'implémentation et d'efficacité.

On notera que la notion de pile pleine n'existe plus, à proprement parler, compte tenu de l'aspect dynamique de ce composant. Par ailleurs, il n'y a plus de raison d'interdire l'affectation.

Voici ce que devient le programme d'essai de l'exercice 42 dans ce cas :

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std ;
```

```
main()
{
    void fct (stack<int, vector<int> >) ;
     stack<int, vector<int> > pile ;  // il faut un conteneur
de base, ici vector
     cout << "vide : " << pile.empty() << "\n" ;  // ici, on
affiche un booléen
          pile.push(1)  ;  pile.push(2)  ;  pile.push(3)  ;
pile.push(4) ;
    fct (pile) ;
    int n, p ;
       n  =  pile.top()  ;  pile.pop()  ;  p  =  "pile.top()  ;
pile.pop() ;  // dépile

   // 2 valeurs
    cout << "haut de la pile au retour de fct : " << n << " "
<< p << "\n" ;
    stack <int, vector<int> > pileb ;
    pileb = pile ;       // ici l'affectation fonctionne !!!
}
void fct (stack <int, vector<int> > pl)
{
    cout << "haut de la pile reçue par fct : " ;
    int  n, p ;
     n = pl.top(); pl.pop() ; p = pl.top() ; pl.pop() ; // on
dépile 2 valeurs
    cout << n << " " << p << "\n" ;
    pl.push(12) ;                                        // on
en ajoute une
}
```

```
vide : 1
haut de la pile recue par fct : 4 3
haut de la pile au retour de fct : 4 3
```

Exercice 158 (144 revisité)

Ancien énoncé

Réaliser une classe nommée `set_int` permettant de manipuler des

ensembles de nombres entiers. Le nombre maximal d'entiers que pourra contenir l'ensemble sera précisé au constructeur qui allouera dynamiquement l'espace nécessaire. On prévoira les opérateurs suivants (`e` désigne un élément de type `set_int` et `n` un entier :

- `<<`, tel que `e<<n` ajoute l'élément `n` à l'ensemble `e` ;
- `%`, tel que `n%e` vale `1` si `n` appartient à `e` et `0` sinon ;
- `<<`, tel que `flot << e` envoie le contenu de l'ensemble `e` sur le flot indiqué, sous la forme :

```
[entier1, entier2, ... entiern]
```

La fonction membre `cardinal` fournira le nombre d'éléments de l'ensemble. Enfin, on s'arrangera pour que l'affectation ou la transmission par valeur d'objets de type `set_int` ne pose aucun problème (on acceptera la duplication complète d'objets).

Commentaires

Si l'on ne s'intéresse qu'aux seules fonctionnalités de la classe, en dehors de la sortie sur un flot, celles-ci sont fournies intégralement par le composant standard `set<int>`. En ce qui concerne la sortie sur un flot, on dispose de plusieurs démarches. On peut bien sûr la programmer au fur et à mesure des besoins, en écrivant à chaque fois les quelques instructions de parcours de l'ensemble :

```
set<int> ens ;
set<int>::iteratos ie ;
   .....
for (ie=ens.begin() ; ie!=ens.end() ; ie++)  cout << *ie << "
" ;
```

On peut aussi en faire une fonction ordinaire, comme dans cet exemple, analogue à l'exemple d'utilisation de l'exercice 146 :

```
#include <iostream>
#include <set>
using namespace std ;

void affiche (set<int>) ;
main()
{  void fct (set<int>) ;
```

```
   void fctref (set<int> &) ;
   set<int> ens ;
   cout << "donnez 10 entiers \n" ;
   int i, n ;
   for (i=0 ; i<10 ; i++)
      { cin >> n ;
        ens.insert(n) ;
      }
      cout << "il y a : " << ens.size() << " entiers
différents\n" ;
   cout << "qui forment l\'ensemble : " ; affiche(ens) ;
   fct (ens) ;
    cout << "au retour de fct, il y en a " << ens.size() <<
"\n" ;
   cout << "qui forment l\'ensemble : " ; affiche(ens) ;
   fctref (ens) ;
   cout << "au retour de fctref, il y en a " << ens.size() <<
"\n" ;
   cout << "qui forment l\'ensemble : " ; affiche(ens) ;
    cout << "appartenance de -1   : " << ens.count(-1)   <<
"\n" ;
    cout << "appartenance de 500 : " << ens.count(500) <<
"\n" ;
   set<int> ensa, ensb ;
   ensa = ensb = ens ;
   cout << "ensemble a : " ; affiche(ensa) ;
   cout << "ensemble b : " ; affiche(ensb) ;
}

void fct (set<int> e)
{  cout << "ensemble reçu par fct : " ; affiche(e) ;
   e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}

void fctref (set<int> & e)
{  cout << "ensemble reçu par fctref : " ; affiche(e) ;
   e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}

void affiche (set<int> e)
{ set<int>::iterator ie ;
  cout << "[ " ;
  for (ie=e.begin() ; ie!=e.end() ; ie++)
     cout << *ie << " ";
  cout <<"] \n" ;
}
```

```
donnez 10 entiers
3 5 3 1 8 5 1 7 7 3
il y a : 5 entiers differents
qui forment l'ensemble : [ 1 3 5 7 8 ]
ensemble recu par fct : [ 1 3 5 7 8 ]
au retour de fct, il y en a 5
qui forment l'ensemble : [ 1 3 5 7 8 ]
ensemble recu par fctref : [ 1 3 5 7 8 ]
au retour de fctref, il y en a 8
qui forment l'ensemble : [ -3 -2 -1 1 3 5 7 8 ]
appartenance de -1  : 1
appartenance de 500 : 0
ensemble a : [ -3 -2 -1 1 3 5 7 8 ]
ensemble b : [ -3 -2 -1 1 3 5 7 8 ]
```

On peut également créer artificiellement une classe `set_int`, dérivée de `set<int>`, dans laquelle on surdéfinit l'opérateur `<<`, comme dans cet exemple qui fournit les mêmes résultats que le précédent :

```
#include <iostream>
#include <set>
using namespace std ;

   /************* déclaration de la classe set_int *********/
class set_int : public set<int>
{ public :
     // envoi ensemble dans un flot
   friend ostream & operator << (ostream &, set_int &) ;
} ;

   /************* définition de la classe set_int *********/
ostream & operator << (ostream & sortie, set_int & e)    //
voir remarque 1
{  sortie << "[ " ;
   set<int>::iterator ie ;
   for (ie=e.begin() ; ie!=e.end() ; ie++)
      sortie << *ie << " " ;
   sortie << "]" ;
   return sortie ;
}

   /************* test de la classe set_int *********/
main()
{
```

```
   void fct (set_int) ;
   void fctref (set_int &) ;
   set_int ens ;
   cout << "donnez 10 entiers \n" ;
   int i, n ;
   for (i=0 ; i<10 ; i++)
      { cin >> n ;
        ens.insert(n) ;
      }
      cout << "il y a : " << ens.size() << " entiers
différents\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fct (ens) ;
    cout << "au retour de fct, il y en a " << ens.size() <<
"\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
   fctref (ens) ;
    cout << "au retour de fctref, il y en a " << ens.size() <<
"\n" ;
   cout << "qui forment l\'ensemble : " << ens << "\n" ;
     cout << "appartenance de -1   : " << ens.count(-1)   <<
"\n" ;
     cout << "appartenance de 500 : " << ens.count(500) <<
"\n" ;
   set_int ensa, ensb ;
   ensa = ensb = ens ;
   cout << "ensemble a : " << ensa << "\n" ;
   cout << "ensemble b : " << ensb << "\n" ;
}

void fct (set_int e)
{  cout << "ensemble reçu par fct : " << e << "\n" ;
   e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}

void fctref (set_int & e)
{  cout << "ensemble reçu par fctref : " << e << "\n" ;
   e.insert(-1) ; e.insert(-2) ; e.insert(-3) ;
}
```

Remarques

1. Certains environnements imposent que l'on mentionne l'espace de nom `std` dans la surdéfinition de l'opérateur << en écrivant `std::operator <<`.

2. Ici, il n'est pas nécessaire de redéfinir l'affectation ou le constructeur par recopie. En effet, compte tenu des règles relatives à l'héritage, les fonctions par défaut appellent bien les fonctions voulues dans la classe de base ; cela suffit ici puisque la classe dérivée n'introduit aucune partie dynamique supplémentaire.

Exercice 159 (145 revisité)

Ancien énoncé

Créer une classe `vect` permettant de manipuler des « vecteurs dynamiques » d'entiers, c'est-à-dire des tableaux d'entiers dont la dimension peut être définie au moment de leur création (une telle classe a déjà été partiellement réalisée dans l'exercice 82). Cette classe devra disposer des opérateurs suivants :

- `[]` pour l'accès à une des composantes du vecteur, et cela aussi bien au sein d'une expression qu'à gauche d'une affectation (mais cette dernière situation ne devra pas être autorisée sur des « vecteurs constants ») ;
- `==`, tel que si `v1` et `v2` sont deux objets de type `vect`, `v1==v2` prenne la valeur `1` si `v1` et `v2` sont de même dimension et ont les mêmes composantes et la valeur `0` dans le cas contraire ;
- `<<`, tel que `flot<<v` envoie le vecteur `v` sur le flot indiqué, sous la forme :

```
<entier1, entier2, ... , entiern>
```

De plus, on s'arrangera pour que l'affectation et la transmission par valeur d'objets de type `vect` ne pose aucun problème ; pour ce faire, on acceptera de dupliquer complètement les objets concernés.

Commentaires

En dehors de la sortie sur un flot, le composant `vector<int>` répond

parfaitement à la question (avec, ici, les mêmes noms d'opérateurs `[]` et ==). En ce qui concerne la sortie sur un flot, on dispose de plusieurs démarches, comme dans l'exercice précédent. On peut bien sûr la programmer au fur et à mesure des besoins, en écrivant à chaque fois les quelques instructions de parcours de l'ensemble :

```
vector<int> v ;
vector<int>::iteratos iv ;
   .....
for (iv=v.begin() ; iv!=v.end() ; iv++)  cout << *iv << " " ;
```

On peut aussi en faire une fonction ordinaire, comme dans cet exemple, analogue à l'exemple d'utilisation de l'exercice 147 :

```
#include <iostream>
#include <vector>
using namespace std ;
main()
{  void affiche (vector<int>) ;
   int i ;
   vector<int> v1(5), v2(10) ;
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " ; affiche(v1) ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " ; affiche(v2);
   v1 = v2 ;
   cout << "v1 = " ; affiche(v1) ;
   vector<int> v3 = v1 ;
   cout << "v3 = " ; affiche(v3) ;
   vector<int> v4 = v2 ;
   cout << "v4 = " ; affiche(v4) ;
    // const vector<int> w(3) ; w[2] = 5 ; // conduit bien à
erreur compilation
}
void affiche (vector<int> v)
{ vector<int>::iterator iv ;
  for (iv=v.begin() ; iv !=v.end() ; iv++)
  cout << *iv << " " ;
  cout << "\n" ;
}
```

```
 v1 = 0 1 2 3 4
 v2 = 0 1 4 9 16 25 36 49 64 81
 v1 = 0 1 4 9 16 25 36 49 64 81
 v3 = 0 1 4 9 16 25 36 49 64 81
 v4 = 0 1 4 9 16 25 36 49 64 81
```

On peut également créer artificiellement une classe `vect`, dérivée de `vector<int>`, dans laquelle on surdéfinit l'opérateur `<<`, comme dans cet exemple qui fournit les mêmes résultats que le précédent :

```
#include <iostream>
#include <vector>
using namespace std ;
class vect : public vector<int>
{ public :
           vect(int   n)   :   vector<int>(n)   {}          //
indispensable !!!!!!!
    friend ostream & operator << (ostream &, vect &) ;
} ;
ostream & operator << (ostream & sortie, vect & v)  // voir
remarque ci-après
{ sortie << "<" ;
  vector<int>::iterator iv ;
   for (iv=v.begin() ; iv!=v.end(); iv++) sortie << *iv << "
" ;
  sortie << ">" ;
  return sortie ;
}

main()
{  int i ;
   vect v1(5), v2(10) ;
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " << v1 << "\n" ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " << v2 << "\n" ;
   v1 = v2 ;
   cout << "v1 = " << v1 << "\n" ;
   vect v3 = v1 ;
   cout << "v3 = " << v3 << "\n" ;
   vect v4 = v2 ;
   cout << "v4 = " << v4 << "\n" ;
    // const vect w(3) ; w[2] = 5 ;  // conduit bien à erreur
compilation
}
```

Remarque

Certains environnements imposent que l'on mentionne l'espace de nom `std` dans la surdéfinition de l'opérateur `<<` en écrivant `std::operator <<`.

Exercice 160 (96 revisité)

Ancien énoncé

Réaliser une classe nommée `bit_array` permettant de manipuler des tableaux de bits (autrement dit, des tableaux dans lesquels chaque élément ne peut prendre que l'une des deux valeurs `0` ou `1`). La taille d'un tableau (c'est-à-dire le nombre de bits) sera définie lors de sa création (par un argument passé à son constructeur). On prévoira les opérateurs suivants :

- `+=`, tel que `t+=n` mette à `1` le bit de rang `n` du tableau `t` ;
- `-=`, tel que `t-=n` mette à `0` le bit de rang `n` du tableau `t` ;
- `[],` tel que l'expression `t[i]` fournisse la valeur du bit de rang `i` du tableau `t` (on ne prévoira pas, ici, de pouvoir employer cet opérateur à gauche d'une affectation, comme dans `t[i] = ...`) ;
- `++`, tel que `t++` mette à `1` tous les bits de `t` ;
- `--`, tel que `t--`mette à `0` tous les bits de `t` ;
- `<<`, tel que `flot << t` envoie le contenu de `t` sur le flot indiqué, sous la forme :

```
<* bit1, bit2, ... bitn *>
```

On fera en sorte que l'affectation et la transmission par valeur d'objets du type `bit_array` ne pose aucun problème.

Commentaires

Si l'on ne s'intéresse qu'aux seules fonctionnalités de la classe qu'on demande d'écrire et que l'on fait abstraction de l'opérateur de sortie sur un flot, le composant standard `vector<bool>` fera l'affaire. On notera cependant qu'alors, l'opérateur `[]` pourra être employé à gauche d'une affectation.

Si l'on tient absolument à ce que ces fonctionnalités soient mises en œuvre par l'intermédiaire des opérateurs proposés, on peut quand même s'appuyer sur les fonctionnalités de la classe `vector<bool>` en créant une classe dérivée qu'on adapte de façon appropriée. Voici ce que pourrait être la définition

d'une telle classe, nommée `bit_array`, accompagnée du même exemple d'utilisation que dans l'exercice 96 :

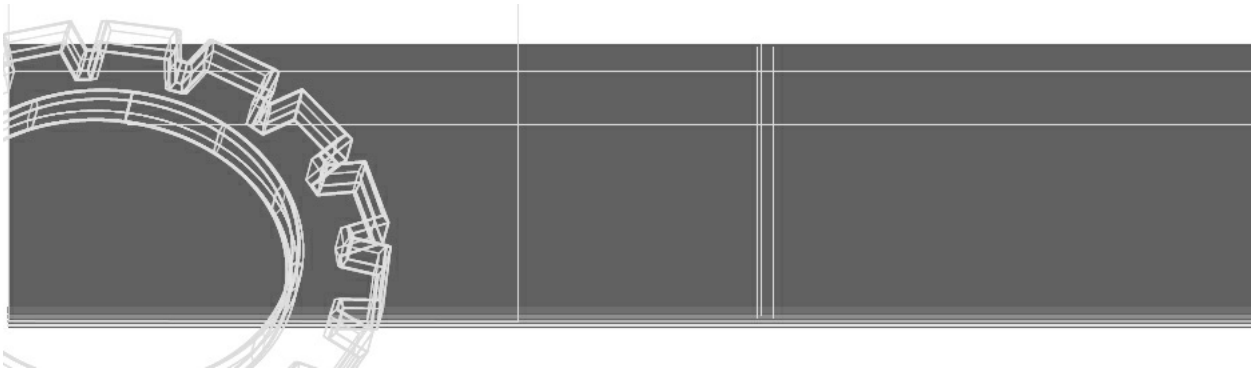
```
#include <iostream>
#include <vector>
#include <limits.h>
using namespace std ;
            /* déclaration de la classe bit_array */
class bit_array : public vector<bool>
{ public :
    bit_array (int = 16) ;
    int operator [] (int) const ;              // valeur d'un
bit
    void operator += (int) ;                    // activation
d'un bit
      void operator -= (int) ;                              //
désactivation d'un // bit
                                                 // envoi sur
flot
    friend ostream & operator << (ostream &, bit_array &) ;
                                                          // les
opérateurs unaires
    void operator ++ () ;                      // mise à 1
    void operator -- () ;                      // mise à 0
    void operator ~ () ;                        // complément
à 1
} ;

 /* définition des fonctions de la classe bit_array */
bit_array::bit_array (int nb): vector<bool> (nb) {}
void bit_array::operator += (int i)
{ (*this).vector<bool>::operator[](i) = true ;
}  // vector<bool>::operator[] pour forcer l'emploi de [] de
classe de base

void bit_array::operator -= (int i)
{ (*this).vector<bool>::operator[](i) = false ;
}  // vector<bool>::operator[] pour forcer l'emploi de [] de
classe de base

ostream & operator << (ostream & sortie, bit_array & t) //
voir remarque
{ sortie << "<* " ;
  vector<bool>::iterator ie ;
  for (ie=t.begin() ; ie!=t.end() ; ie++)
    sortie << *ie << " " ;
  sortie << "*>" ;
```

```
  return sortie ;
}

void bit_array::operator ++ ()
{ vector<bool>::iterator ie ;
  for (ie=(*this).begin() ; ie!=(*this).end() ; ie++)
    *ie = true ;
}

void bit_array::operator -- ()
{ vector<bool>::iterator ie ;
  for (ie=(*this).begin() ; ie!=(*this).end() ; ie++)
    *ie = false ;
}

void bit_array::operator ~ ()
{ vector<bool>::iterator ie ;
  for (ie=(*this).begin() ; ie!=(*this).end() ; ie++)
    *ie = !(*ie) ;
}
   /* programme d'essai de la classe bit_array */
main ()
{ bit_array t1 (34) ;
  cout << "t1 = " << t1 << "\n" ;
  t1 +=3 ; t1 += 0 ; t1 +=8 ; t1 += 15 ; t1 += 33 ;
  cout << "t1 = " << t1 << "\n" ;
  t1-- ;
  cout << "t1 = " << t1 << "\n" ;
  t1++ ;
  cout << "t1 = " << t1 << "\n" ;
  t1 -= 0 ; t1 -= 3 ; t1 -= 8 ; t1 -= 15 ; t1 -= 33 ;
  cout << "t1 = " << t1 << "\n" ;
  cout << "t1 = " << t1 << "\n" ;
  bit_array t2 (11), t3 (17) ;
  cout << "t2 = " << t2 << "\n" ;
  t2 = t3 = t1 ;
  cout << "t3 = " << t3 << "\n" ;
}
```

Certains environnements imposent que l'on mentionne l'espace de nom `std` dans la surdéfinition de l'opérateur `<<` en écrivant `std::operator <<`.

La comparaison avec l'exercice correspondant du chapitre 16 montre que le gain obtenu avec l'utilisation des composants standard reste assez relatif. L'essentiel vient de ce qu'il n'est plus besoin ici de surdéfinir l'opérateur d'affectation et le constructeur par recopie.

Chapitre 22
Les pointeurs intelligents (C++11/C++14)



Rappels

Le pointeur null

C++11 a introduit la notation `nullptr` représentant une valeur (de type `nullptr_t`) :

- convertible implicitement en un pointeur de n'importe quel type ;
- pouvant être comparée à un pointeur de n'importe quel type.

Les pointeurs intelligents

C++11 a introduit de nouveaux types patrons dits « pointeurs intelligents », destinés à combler certaines lacunes des « pointeurs nus » lorsqu'on les utilise dans le cadre de la gestion dynamique de mémoire. Nous ne nous

intéresserons ici qu'aux deux principaux `unique_ptr` et `shared_ptr` destinés à gérer de façon automatique **l'emplacement pointé qui sera automatiquement détruit lorsqu'il ne sera plus référencé** (le recours à *delete* est inutile et interdit). Ces deux types ne sont pas adaptés aux références circulaires.

Le type unique_ptr

Le type `unique_ptr<T>` est conçu de manière que, à un instant donné, un emplacement mémoire donné ne soit référencé que par un seul pointeur de ce type, considéré comme le « propriétaire » de l'emplacement pointé. Il peut être déclaré :

- sans être initialisé comme dans (`T` étant un type classe) :

```
unique_ptr <T> ap1 ;   // ap1 vaut alors nullptr
```

- en fournissant à son constructeur une expression de même type `unique_ptr<T>` (ou `T*`) ou en l'initialisant avec une expression de même type `T` (ou de type `T*`) comme dans (..... désignant des arguments pour un constructeur de `T`) ;

```
   unique_ptr <T> ap2 (new T(.....)) ;           // avec C++11
   unique_ptr <T> ap2 = make_unique <T>(.....) ; // préférable
depuis
                     // C++14, car mieux protégé vis-à-vis des
exceptions
```

Le transfert de propriété d'un emplacement à un autre pointeur du même type ne peut pas se faire par simple affectation, laquelle sera rejetée en compilation ; il faut recourir à la fonction `move` de cette façon :

```
unique_ptr <T> ap2 (new T(.....)) ;
ap1 = move(ap2) ;  // ap1 prend la valeur de ap2 et devient
propriétaire
                   // ap2 n'est plus valide (vaut nullptr)
```

De même, si `f` est une fonction d'en-tête :

```
f(unique_ptr<T> ap)
```

L'appel `f(ap2)`, qui provoquerait une copie de `ap2`, sera interdit. Là encore, il faudra recourir à la fonction `move` de cette façon :

```
f(move(ap2)) ;   // l'argument ap de f prend la valeur de ap2
et devient
```

```
                    // propriétaire ; ap2 n'est plus valide (vaut
  nullptr)
```

La fonction membre `get` permet de retrouver la valeur effective du pointeur : `ap1.get()` représente la valeur du pointeur (de type `double*`) contenu dans `ap1`.

Le type shared_ptr

Le type `shared_ptr<T>` permet quant à lui, à un même emplacement, d'être référencé par plusieurs pointeurs. Il peut lui aussi être déclaré sans initialisation ou en fournissant à son constructeur une expression de même type (ou de type `T*`) ou en l'initialisant avec une expression de même type (ou de type `T*`) :

```
shared_ptr <T> ap1 ;    // ap1 vaut nullptr
shared_ptr <T> ap2 = make_shared <T> (...) ;
 // préférable à :  shared_ptr <T> ap2 (new T(...)) ;
```

Un mécanisme approprié associe à un emplacement donné un « compteur de références », qui comptabilise le nombre de références (de type `shared_ptr`) portant sur cet emplacement. On peut en connaître la valeur en utilisant la fonction membre `use_count`. La fonction membre `unique` vaut `true` lorsque l'emplacement est référencé par un seul pointeur et `false` sinon. L'emplacement en question est automatiquement libéré lorsque le compteur de références correspondant tombe à zéro.

Cette fois, l'affectation et la copie de pointeurs sont autorisées (notamment lors de la transmission par valeur en argument d'une fonction). Naturellement, le compteur de références est actualisé en conséquence.

La fonction membre `use_count` fournit le nombre d'emplacements référencés, tandis que la fonction membre `unique` vaut `true` lorsque l'emplacement est référencé par un seul pointeur et `false` sinon.

Là encore, la fonction membre `get` permet de retrouver la valeur effective du pointeur.

---

1. Il est possible de transmettre un pointeur intelligent par référence (à condition qu'il ne s'agisse pas d'un pointeur déclaré constant). Dans ce

cas, cela ne modifie pas le nombre de pointeurs référençant un emplacement donné.

2. Le type `unique_ptr` (mais pas la fonction `make_unique`) peut s'appliquer à des tableaux qu'on utilise classiquement avec l'opérateur `[]`, en procédant ainsi :

```
   unique_ptr<double[]> adr ;   // adr est un unique_ptr sur
un tableau
                                // d'éléments de type double
```

3. On peut appliquer `move` à un `shared_ptr` (`ap1` et `ap2` étant de ce type), comme dans :

```
ap1 = move(ap2) ;
```

Dans ce cas, `ap1` reçoit bien une copie de `ap2`, mais `ap2` est invalidé (`nullptr`). La même remarque s'appliquerait à la transmission par valeur en argument d'une fonction.

4. Nous verrons dans le chapitre suivant que le rôle précis de `move` est de convertir son argument en `rvalue` et de forcer ainsi le recours à des opérations (affectation ou copie) dites « de déplacement ».

Exercice 161

Énoncé

a. Quelles erreurs ont été commises dans le programme suivant ?

```
#include <iostream>
#include <memory>
using namespace std ;
int main()
{  void f(unique_ptr<int>) ;
   unique_ptr<int> pi = make_unique <int>(5) ;
   f(pi) ;
   cout << "valeur = " << *pi << endl ;
}
void f(unique_ptr<int> p)
{ cout << "valeur = " << *p << endl ;
```

```
}
```

b. Que se passerait-il si l'on remplaçait le type `unique_ptr` par `shared_ptr` ?

Solutions

a.

Dans l'appel :

```
f(pi) ;
```

`pi` doit être transmis par valeur, ce qui en implique une copie ; l'appel sera rejeté en compilation, car on ne peut pas copier une valeur de type `unique_ptr`.

On peut le remplacer par :

```
f(move(pi)) ;
```

ce qui transmet une copie de `pi` à `f`, en invalidant le pointeur `pi`. Dans ce cas, la fonction `f` affiche bien la valeur de `*p` (c'est-à-dire `5`). En revanche, la dernière instruction :

```
cout << "valeur = " << *pi << endl ;
```

conduirait à une erreur d'exécution due à ce que la valeur de `p` est nulle et qu'il n'y a donc plus d'objet pointé.

b.

Avec le type `shared_ptr`, les copies sont autorisées, de sorte que l'instruction :

```
f(pi) ;
```

sera bien acceptée en compilation.

Le programme fonctionnera correctement et la dernière instruction affichera bien à son tour la valeur de `*p` (c'est-à-dire `5`).

Exercice 162

Énoncé

Quels seront les résultats fournis par ce programme :

```
#include <iostream>
#include <memory>
using namespace std ;
void f(shared_ptr<int> p)
{  cout  <<  "---Entree  f    p  "  <<  *p  <<  "  "    <<
p.use_count()   << endl ;
  shared_ptr <int> q (p);
    cout  <<  "---Sortie  f    p  "  <<  *p  <<  "  "    <<
p.use_count()   << endl ;
    cout  <<  "---Sortie  f    q  "  <<  *q  <<  "  "    <<
q.use_count()   << endl ;
}
int main()
{ shared_ptr <int> pi1 = make_shared <int> (10) ;
    cout  <<  "--En  1         pi1  "  <<  *pi1  <<  "  "  <<
pi1.use_count() << endl ;
  f(pi1) ;
    cout  <<  "--En  2         pi1  "  <<  *pi1  <<  "  "  <<
pi1.use_count() << endl ;
  shared_ptr <int> pi2 = make_shared <int> (20) ;
    cout  <<  "--En  2         pi2  "  <<  *pi2  <<  "  "  <<
pi2.use_count() << endl ;
  shared_ptr<int> pi3(pi2) ;
    cout  <<  "--En  3         pi2  "  <<  *pi2  <<  "  "  <<
pi2.use_count() << endl ;
    cout  <<  "--En  3         pi3  "  <<  *pi3  <<  "  "  <<
pi3.use_count() << endl ;
  pi1 = pi2 ;
    cout  <<  "--En  4         pi1  "  <<  *pi1  <<  "  "  <<
pi1.use_count() << endl ;
    cout  <<  "--En  4         pi2  "  <<  *pi2  <<  "  "  <<
pi2.use_count() << endl ;
    cout  <<  "--En  4         pi3  "  <<  *pi3  <<  "  "  <<
pi3.use_count() << endl ;
  pi3 = pi2 ;
    cout  <<  "--En  5         pi2  "  <<  *pi2  <<  "  "  <<
pi2.use_count() << endl ;
    cout  <<  "--En  5         pi3  "  <<  *pi3  <<  "  "  <<
pi3.use_count() << endl ;
}
```

Solutions

Un appel de fonction tel que `f(p1)` crée une copie de `p1`, ce qui augmente de 1 le compteur de références associé.

Par ailleurs, une affectation telle que :

```
pi1 = pi2 ;
```

conduit à diminuer de 1 le compteur de références de l'objet pointé par `pi2` et à augmenter de 1 celui de l'objet pointé par `pi1`. Pour mettre cela en évidence, il faut afficher le compteur de références de l'ancien objet pointé par `pi2`, à savoir ici l'objet pointé par `pi3`. On constate bien que ce dernier est alors passé de `2` à `3`.

Enfin, dans l'affectation :

```
pi3 = pi2 ;
```

les deux pointent déjà sur le même objet, de sorte que l'affectation ne fait rien.

En définitive, voici les résultats obtenus (auxquels nous avons ajouté quelques commentaires introduits par //) :

```
--En 1     pi1 10 1        // création pi1
---Entree f  p 10 2        // copie pi1 dans p de f
---Sortie f  p 10 3        // création q copie de p
---Sortie f  q 10 3
--En 2     pi1 10 1        // p et q de f ont été supprimés
--En 2     pi2 20 1
--En 3     pi2 20 2        // création pi3 copie de pi2
--En 3     pi3 20 2
--En 4     pi1 20 3        // pi1 = pi2
--En 4     pi2 20 3
--En 4     pi3 20 3
--En 5     pi2 20 3        // pi3 = pi2 ne fait rien
--En 5      pi3 20 3          // car pi2 et pi3 ont la même
valeur
```

Exercice 163

Énoncé

Quels seront les résultats fournis par ce programme :

```
#include <iostream>
#include <memory>
using namespace std ;
void affiche (shared_ptr<int> p)
{ if (p != nullptr)
    { cout << "-- Dans affiche - valeur = " << *p
           << " compte = "  << p.use_count()   << endl ;
    }
  else cout << "-- Dans affiche, p est nul\n" ;
}
void ajoute (shared_ptr<int> p, int n)
{ if (p != nullptr) *(p) += n ;
     else cout << "-- Dans ajoute, p est nul\n" ;
}
int main()
{ shared_ptr<int> pi1 = make_shared <int> (10) ;
  cout <<  "-- En I  pi1    - valeur = " << *pi1
       << " compte = " << pi1.use_count() << endl ;
  cout << "-- En II\n" ;
  affiche(pi1) ;
  ajoute (pi1, 5) ;
  affiche(pi1) ;
  shared_ptr<int> pi2(pi1) ;
  cout << "-- En III\n" ;
  ajoute (pi2, 25) ;
  affiche(pi1) ;
  cout << "-- En IV\n" ;
  affiche (move(pi1));
  ajoute (pi1, 3 ) ;
  affiche (pi1);
}
```

Solutions

Ici, la fonction `affiche` reçoit une copie d'un `shared_ptr` (et non une référence), de sorte que le compteur associé à l'entier pointé se trouve incrémenté de 1 à l'entrée dans la fonction. D'où la différence constatée entre les affichages du compteur de `pi1` dans `main` et dans `affiche`.

En revanche, cela ne nuit nullement au fonctionnement de `ajoute` puisqu'il

accède bien en l'entier voulu à l'aide de la copie du pointeur.

La création de `pi2` se fait par copie de `pi1`, de sorte que les deux pointeurs désignent alors le même entier. On constate qu'on a pu l'incrémenter de 25 en utilisant `pi2` et l'afficher en utilisant `pi1`. `affiche(move(pi1))` (rappelons que `move` s'applique aussi à un `shared_ptr`) appelle `affiche` en lui transmettant une copie de `pi1`, avant d'invalider ce dernier.

En définitive, voici les résultats obtenus :

```
-- En I  pi1    - valeur = 10 compte = 1
-- En II
-- Dans affiche - valeur = 10 compte = 2
-- Dans affiche - valeur = 15 compte = 2
-- En III
-- Dans affiche - valeur = 40 compte = 3
-- En IV
-- Dans affiche - valeur = 40 compte = 2
-- Dans ajoute, p est nul
-- Dans affiche, p est nul
```

Lorsque vous aurez étudié la sémantique de déplacement présentée dans le chapitre suivant, vous verrez que `move` convertit son argument `pi1` en une `rvalue`, de sorte que le constructeur de copie utilisé dans `f(move(pi1))` est le constructeur de copie par déplacement. En revanche, une instruction expression telle que :

```
move(pi2) ;
```

n'aurait pas plus d'effet que l'instruction expression :

```
double(n) ;
```

Exercice 164

Énoncé

Que se passerait-il si, dans le programme de l'exercice précédent, les fonctions `affiche` et `ajoute` recevaient leurs arguments de type `shared_ptr` par référence, de la manière suivante :

```
void affiche (const shared_ptr<int> &  p)
void ajoute  (shared_ptr<int> & p, int n)
```

Solution

En transmettant les pointeurs par référence, on pourra accéder dans `affiche` à la bonne valeur du compteur de références puisqu'il n'aura plus été modifié. Voici les résultats obtenus :

```
-- En I  pi1    - valeur = 10 compte = 1
-- En II
-- Dans affiche - valeur = 10 compte = 1
-- Dans affiche - valeur = 15 compte = 1
-- En III
-- Dans affiche - valeur = 40 compte = 2
-- En IV
-- Dans affiche - valeur = 43 compte = 2
-- Dans affiche - valeur = 43 compte = 2
```

Remarque

Comme vous le verrez plus en détail dans le prochain chapitre, l'instruction :

```
affiche (move(pi1));
```

n'est acceptée en compilation que parce que l'on a prévu une référence à une constante dans `affiche`, car l'expression `move(pi1)` est une `rvalue` et non une `lvalue`.

Exercice 165

Énoncé

On dispose de la classe `point` suivante :

```
class point
{ int x, y ;
 public :
  point (int abs, int ord) { x = abs ; y = ord ; }
  int get_x() { return x ; }
  int get_y() { return y ; }
} ;
```

On se propose d'écrire une classe `segment`, disposant ici uniquement d'un constructeur à deux arguments (pointeurs sur chacun des points correspondant aux extrémités du segment) et d'une fonction membre `longueur` affichant la longueur du segment correspondant.

a. Écrire cette classe dans le cas où les deux points du segment sont désignés par des pointeurs conventionnels (`point *`). Écrire un petit programme l'utilisant :

- sur des points automatiques (déclaration usuelle) ;
- sur des points dynamiques dont l'adresse figure dans un pointeur conventionnel (de type `point *`) ;
- sur des points dynamiques dont l'adresse figure dans un pointeur de type `shared_ptr`.

b. Écrire cette même classe dans le cas où les deux points du segment sont désignés par des pointeurs de type `shared_ptr`. Est-il encore possible d'écrire un programme l'utilisant dans les trois cas évoqués ?

Solutions

a.

L'écriture de la classe `segment` est « classique »

```
#include <cmath> // pour sqrt
class segment
{ point * ad_debut, * ad_fin ;
 public :
  segment (point * add, point * adf)
  { ad_debut = add ; ad_fin = adf ;
  }
  float longueur()
```

```
  { int dx = (*ad_fin).get_x() - (*ad_debut).get_x()  ;
    int dy = (*ad_fin).get_y() - (*ad_debut).get_y()  ;
    return sqrt (dx*dx + dy*dy) ;
  }
} ;
```

Son utilisation avec des adresses ou pointeurs conventionnels est, elle aussi, classique. Seule l'utilisation avec des pointeurs de type `shared_ptr` nécessite de retrouver le pointeur nu de type `point *` qu'il encapsule. On utilise pour cela la méthode `get`.

```
#include <iostream>
#include <memory>
int main()
{ point a (1, 5) ;
  point b (3, 2) ;
     // cas 1 : points automatiques
  segment s1 (&a, &b) ;
  cout << "longueur s1 = " << s1.longueur() << endl ;
     // cas 2 : points repérés par point *
  point * adp1 = new point (3, 9);
  point * adp2 = new point (1, 4) ;
  segment s2 (adp1, adp2) ;
  cout << "longueur s2 = " << s2.longueur() << endl ;
     // cas 3 : points repérés par des shared_ptr <point>
  shared_ptr <point> sp1 = make_shared <point> (1, 3);
  shared_ptr <point> sp2 = make_shared <point> (2, 4);
  segment sh (sp1.get(), sp2.get() ) ;
  cout << "longueur sh = " << sh.longueur() << endl ;
}

longueur s1 = 3.60555
longueur s2 = 5.38516
longueur sh = 1.41421
```

b.

Notre classe `segment` pourrait se présenter ainsi :

```
#include <memory>
class segment
{ shared_ptr <point> ad_debut, ad_fin ;
 public :
  segment (shared_ptr <point> add, shared_ptr<point> adf)
  { ad_debut = add ; ad_fin = adf ;
  }
  float longueur()
  { int dx = (*ad_fin).get_x() - (*ad_debut).get_x()  ;
```

```
    int dy = (*ad_fin).get_y() - (*ad_debut).get_y()  ;
    return sqrt (dx*dx + dy*dy) ;
  }
} ;
```

On pourrait naturellement penser à traiter les trois situations évoquées en procédant ainsi :

```
#include <iostream>
#include <memory>
int main()
{ point a(1, 5) ;
  point b (3, 2) ;
      // cas 1 : points automatiques -> erreur en fin de main
  shared_ptr <point> ada (&a) ;
  shared_ptr <point> adb (&b) ;
  segment s1 (ada, adb) ;
  cout << "longueur s1 = " << s1.longueur() << endl ;
      // cas 2 : points repérés point *  ->  OK mais risque
  point * adp1 = new point (3, 9);
  point * adp2 = new point (1, 4) ;
  shared_ptr <point> ap1 (adp1) ;
  shared_ptr <point> ap2 (adp2) ;
  segment s2 (ap1, ap2) ;
  cout << "longueur s2 = " << s2.longueur() << endl ;
      // cas 3 : points repérés par des shared_ptr<point>  ->
OK
  shared_ptr<point> sp1 = make_shared <point> (2, 4) ;
  shared_ptr<point> sp2 = make_shared <point> (1, 5) ;
  segment sh (sp1, sp2) ;
  cout << "longueur sh = " << sh.longueur() << endl ;
}

longueur s1 = 3.60555
longueur s2 = 5.38516
longueur sh = 1.41421
```

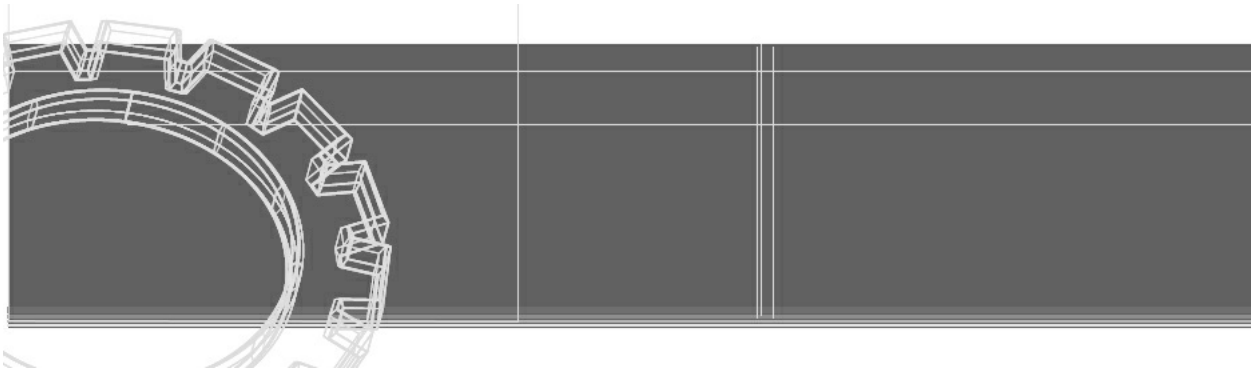
En fait, seule la troisième situation est réaliste. En effet, dans la première, à la fin du programme `main`, les compteurs de références de `sp1` et `sp2` vont tomber à zéro et l'on cherchera à supprimer l'entier pointé, lequel n'est en fait pas un objet dynamique ; on obtiendra probablement une erreur d'exécution (après la fin du programme toutefois). Les choses seraient plus catastrophiques si l'on se trouvait dans une fonction autre que `main` puisqu'alors l'erreur se produirait dès la sortie de cette fonction.

La deuxième situation fonctionne ici, mais elle comporte quand même des

risques dus à ce que l'on se réfère à un même emplacement à la fois par un pointeur conventionnel et par un pointeur intelligent qui ne compte que les références qu'il connaît. Par exemple, dans un gros programme, on court l'un des risques suivants :

- exécuter la fonction `delete` sur le pointeur conventionnel alors que le compteur de références du pointeur intelligent correspondant n'est pas tombé à zéro ; une tentative d'accès via ce pointeur intelligent conduira à une erreur d'exécution ;
- au contraire, le compteur de références peut se trouver nul, entraînant la destruction de l'objet correspondant, alors qu'on peut tenter d'y accéder avec le pointeur conventionnel.

Chapitre 23
La sémantique de déplacement (C++11/C++14)



Rappels

Notion de référence à une rvalue

Parfois, lors d'une contruction par recopie ou lors d'une affectation, l'objet à copier est temporaire (on sait qu'il ne servira plus par la suite). On pourrait donc profiter de cette propriété pour **optimiser le code**, par exemple, en ne recopiant pas les parties dynamiques et en se contentant d'actualiser convenablement les pointeurs correspondants. Pour permettre de prendre en compte une telle situation, C++11 introduit ce qu'il nomme une nouvelle « sémantique de déplacement » (on dit aussi « de mouvement »). Pour ce faire, il définit un nouveau type de référence, à savoir « référence à une `rvalue` » qui se note avec `T&&` (`T` étant un type quelconque), comme dans :

```
void f (T && t)
```

La fonction `f` pourra alors être appelée en lui fournissant une `rvalue` ; il pourra s'agir :

- d'une expression de type `T` dont la fonction recevra l'adresse de l'emplacement ;
- de la valeur de retour d'une fonction fournissant un résultat de type `T`, comme dans `f(g(.....))` ; là encore, l'adresse du résultat temporaire de `g` sera directement transmise à `f` ;
- d'une variable littérale de type `T` lorsque `T` est un type simple ; dans ce cas la valeur sera **recopiée** dans un emplacement temporaire dont `f` recevra l'adresse.

Les règles de choix d'une fonction surdéfinie ont été adaptées pour tenir compte de cette nouveauté, de sorte qu'on peut maintenant définir simultanément :

```
void f (T &t) ;           // I : applicable à une lvalue
modifiable,
void f (const T &t) ; // II : applicable à une lvalue non
modifiable
                        //       ou une lvalue modifiable si I
absent
                      //      ou une rvalue si III absent
void f (T &&t) ;      // III : applicable à une rvalue
```

Notez que la définition simultanée de `f(T&&)` et de `f(T)` crée une ambigüité.

Il est possible de convertir une `lvalue` en une `rvalue`, pour forcer l'emploi d'une fonction recevant une référence sur une `lvalue`. On utilise à cet effet une fonction standard nommée `move` : si `a` est une `lvalue` de type `T`, l'expression `move(a)` (en fait équivalente à `(static_ cast <T&&>(a))` devient une `rvalue` de type `T`. Ainsi, en supposant que les trois formes de fonctions `f` ci-dessus sont définies :

```
f(a) ;           // appelle la forme I,   qui reçoit l'adresse
de a
f(move(a)) ;     // appelle la forme III, qui reçoit l'adresse
de a
```

Lorsqu'une `rvalue` possède un nom (comme `t` dans la déclaration III de `f`), ce nom redevient une `lvalue` : ce serait le cas de `t` dans le corps de `f`. En revanche, une `rvalue` sans nom (comme la valeur de retour désignée par un

appel de fonction), reste bien une rvalue.

Remarques

1. L'instruction `f(move(a))` autorise l'appel d'une fonction qui considérera `a` comme une `rvalue`, c'est-à-dire un temporaire qu'elle pourra utiliser (et modifier) à sa guise. Il est donc prudent d'éviter d'utiliser `a` après cet appel. De plus, si `a` est un objet comportant des parties dynamiques allouées par son constructeur (ce qui sera généralement le cas), il faut éviter de les détruire dans `f`, car elles le seront également par le destructeur de `a`.
2. Bien que cela soit peu usité, il est théoriquement possible de déclarer qu'une fonction renvoie une référence à une `rvalue`, comme dans :

```
int && f (....) ;
```

3. Il est théoriquement possible de prévoir un argument de la forme `const T &&`. Cela ne présente en fait guère d'intérêt, dans la mesure où l'utilisation d'une référence à une `rvalue` est précisément de pouvoir profiter de l'aspect temporaire de l'élément correspondant et, partant, de le modifier.

Application au constructeur et à l'affectation

Grâce à cette nouvelle notion de référence à une `rvalue`, il devient possible de doter une classe d'un constructeur de déplacement, ainsi que d'un opérateur d'affectation par déplacement, en leur prévoyant un argument correspondant à une référence à une `rvalue`. On trouvera donc généralement pour les objets dits alors « déplaçables » :

```
A (const A &) ;              // constructeur de recopie
A (A &&)  ;                  // constructeur par déplacement
```

et :

```
A& operator= (const A &) ;   // affectation par copie
A& operator= (A &&) ;        // affectation par déplacement
```

Il faut bien voir qu'il est alors de la responsabilité du programmeur d'utiliser convenablement l'objet reçu en argument. En particulier, il peut en modifier

les valeurs ou réutiliser des parties dynamiques comme il le souhaite, puisqu'il sait que cet objet ne sera plus utilisé par la suite. Néanmoins, **il devra prendre garde à ce que cet objet reste dans un état acceptable permettant sa destruction ultérieure**, sans préjudice pour le nouvel objet créé par le déplacement.

Il existe des opérations de déplacement par défaut (constructeur et affectation) qui mettent en œuvre des déplacements qu'on pourrait qualifier de « superficiels » puisqu'ils ne s'appliquent pas aux objets pointés. Bien entendu, en cas d'objets membre, il y a récursion sur les opérations de déplacement correspondantes (par défaut ou explicites). Comme on peut s'y attendre, lorsque de telles opérations par défaut sont appliquées à des objets ne comportant pas de parties dynamiques (ou, a fortiori, à des types simples), elles jouent le même rôle que les opérations de copie par défaut.

Ces opérations de déplacement par défaut ne sont pas générées, dès lors que l'on a défini explicitement au moins une des opérations suivantes :

- opération de copie (construction ou affectation) ;
- opération de déplacement (construction ou affectation) ;
- destruction.

On ne perdra pas de vue que, autant la redéfinition des opérations de copie était indispensable avec des classes à parties dynamiques, autant les opérations de déplacement n'ont d'intérêt que pour l'optimisation du code.

Référence universelle dans un patron de fonction

Dans un patron de fonction, si `T` est un paramètre de type, la notation `T&&` (et uniquement celle-là) désigne un paramètre qui peut correspondre indifféremment à une référence à une `lvalue` ou à une référence à une `rvalue`. On la nomme souvent « référence universelle ».

```
void f(int && n)                    { ..... }  // n est une réf
rvalue

template <class T> void g (T&& n) { ..... }  // n est une réf
universelle

   .....

int p = 5 ;
```

```
f(5) ;        // accepté
f(p);         // rejeté car f attend une rvalue
g(5) ;         // OK -->  g(T&&)
g(p) ;         // OK  --> g(T&&)  (si g(T&) pas défini)
```

La fonction forward

La notion de référence universelle induit un nouveau problème dans le cas d'un patron de fonction qui appelle une autre fonction en lui retransmettant l'un de ses arguments, déclaré référence à une `rvalue T&&`. Dans ce cas, la règle de transformation en `lvalue` de toute valeur nommée fait perdre le caractère de référence universelle. La fonction `forward<T>` permet de régler le problème en conservant le caractère de référence universelle de l'argument :

```
template< class T > void essai ( T&& n )
{ f (forward <T> (n) ) ;    // f reçoit une rvalue ou une
lvalue suivant
                            // la nature de n transmise à essai
  f(n) ;                    // f reçoit toujours une lvalue
}
```

À propos des exercices de ce chapitre

On notera bien que certains des exercices proposés ici ne sont destinés qu'à vous faire appliquer les règles relatives à la sémantique de déplacement. Ils utilisent alors soit des types simples, soit des types classe ne comportant pas de parties dynamiques, situations dans lesquelles la sémantique de déplacement ne présente généralement pas d'intérêt.

Exercice 166

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
```

```
using namespace std;
void f(int & n)         { cout << "-- Apel f(int &)  " <<
endl ; }
void f(const int & n)  { cout << "-- Apel f(const int &) "
<< endl ; }
void f(int && n)        { cout << "-- Apel f(int &&) " <<
endl ; }
int g1()    { return 25 ; }
int main()
{
    int n = 10 ;
    const int p = 5 ;
    f (5) ;         // appel 1
    f (n) ;         // appel II
    f (g1()) ;      // appel III
    f (p) ;         // appel IV
    f (2*n + 1) ;   // appel V
}
```

Solutions

Dans l'appel I, `5` est une `rvalue` ; on appelle bien `f(int&&)`, après création d'une copie (car il s'agit d'une constante littérale qu'il n'est pas question de modifier). Notez qu'en l'absence de définition de cette fonction, on aurait appelé `f(const int &)`, là aussi après création d'une copie.

Dans l'appel II, `n` est une `lvalue`, d'ou l'appel de `f(int&)`.

Dans l'appel III, `g1()` représente la valeur de retour de `g1` ; il s'agit bien d'une `rvalue` dont l'adresse sera transmise à `f(int&&)`. Notez qu'il n'y a pas de copie ici.

Dans l'appel IV, `p` est une `lvalue` non modifiable. Elle ne peut correspondre qu'à `f(const int &)`. En l'absence de définition de cette fonction, on obtiendrait une erreur de compilation.

Enfin, dans l'appel V, l'expression `2*n+1` est bien une `rvalue`, d'où l'appel de `f(int &&)` à laquelle on fournit l'adresse de l'emplacement temporaire contenant la valeur de l'expression.

En définitive, on obtient les résultats suivants :

```
-- Apel f(int &&)
-- Apel f(int &)
```

```
-- Apel f(int &&)
-- Apel f(const int &)
-- Apel f(int &&)
```

Remarques

1. Si `g1` avait été défini ainsi :

   ```
   int & g1() { return 25 ; }
   ```

   nous aurions obtenu une erreur de compilation, due à ce que la constante temporaire `25` ne peut pas être renvoyée par référence (alors qu'elle pourra être considérée comme une `rvalue`).

2. Nous aurions pu être tentés de définir `g1` de cette manière :

   ```
   int && g1() { return 25 ; }
   ```

   Mais cela n'est pas utile, puisque la norme a bien prévu que la valeur de retour de `g1` puisse correspondre directement (sans copie) à une référence à une `rvalue`.

Exercice 167

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std;
void f(int & n)          { cout << "-- Apel f(int &)   " <<
endl ; }
void f(const int & n)  { cout << "-- Apel f(const int &) "
<< endl ; }
void f(int && n)         { cout << "-- Apel f(int &&) " <<
endl ; }
int g1()    { return 25 ; }
int main()
{
    int n = 10 ;
    f (move((2*n + 1))) ;   // appel I
```

```
    f (move(g1())) ;        // appel II
    f (move(n)) ;           // appel IIII
}
```

Solutions

En I et II, la fonction `move` ne sert à rien puisque les arguments sont déjà des `rvalue`. Les résultats sont donc les mêmes que si l'on avait écrit :

```
f (2*n + 1) ;
f (g1() ;
```

En I, la fonction `move` convertit `n` en une `rvalue`, d'où l'appel de `f(int &&)`.

En définitive, on obtient ces résultats :

```
-- Apel f(int &&)
-- Apel f(int &&)
-- Apel f(int &&)
```

Remarques

1. On notera bien qu'il s'agit ici encore d'un cas d'école. Notamment, la fonction `f(int&&)` ne « profite pas vraiment » du fait qu'elle reçoit une valeur temporaire qu'elle pourrait éventuellement modifier à volonté (ce qui sera souvent le cas lorsque cette valeur correspondra à un objet à parties dynamiques). Dans ce cas, il serait imprudent de continuer à utiliser `n` dans la suite de la fonction `main`.

2. Si l'on avait trouvé ces instructions :

   ```
   const int q = 5 ; f(move(q) ;
   ```

   dans l'appel de `f`, `move(q)` serait une `lvalue` non modifiable (`const int`) que la fonction `move` convertirait, non pas en une `rvalue`, mais une `rvalue` non modifiable, laquelle ne peut pas correspondre à une référence du type `int &&`. Ici, elle correspondrait donc à `f(const int &)`. En théorie, il reste possible de définir, en plus des 3 fonctions `f` ici présentes, une fonction `f(const int &&)` qui se trouverait donc appelée. Rappelons toutefois qu'une telle définition ne présente généralement pas d'intérêt puisque les références à des `rvalue` sont destinées à indiquer qu'on peut profiter de l'aspect temporaire d'un

objet pour en modifier certaines parties.

Exercice 168

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std;
int h (int & n)    { cout << "---- Appel h(int &) \n" ;
return n ; }
int h (int && n)   { cout << "---- Appel h(int &) \n" ;
return n ; }
void g(int & n)   { cout << "---- Appel g(int &}" << endl ;
}
void g(int && n)  { cout << "---- Appel g(int &&}" << endl ;
}
void f(int && n)
{ cout << "-- Appel f(int &&)\n" ;
  g(n) ;
  g(h(n)) ;
}
void f(int & n)
{ cout << "-- Appel f(int &)\n" ;
  g(n) ;
  g(h(n)) ;
}
int main()
{ cout << "En I\n" ;
  f(10) ;               // appel I
  int n = 20 ;
  cout << "En II\n" ;
  f(n) ;                // appel II
  cout << "En III\n" ;
  f(move(n)) ;          // appel III
}
```

Solutions

Dans l'appel I, `f(10)`, `10` est une `rvalue`, ce qui provoque l'appel de `f(int&& n)` qui reçoit l'adresse d'une copie d'une constante littérale `10`. Mais, dans l'appel `g(n)` de `f`, `n` est redevenu une `lvalue`, ce qui justifie que, parmi les 2 possibles, la fonction `g` appelée soit `g(int&)`. Dans l'appel `g(h(n))`, `n` est une `lvalue`, mais `h(n)` est une `rvalue`, d'où l'appel de `g(int&&)`.

En revanche, dans l'appel II, `f(n)`, `n` est une `lvalue`, ce qui provoque l'appel de `f(int & n)`. Mais, dans le corps de `f`, `n` représente toujours une `lvalue`, de sorte que l'on a affaire aux mêmes appels de `g` et de `h` que précédemment.

Enfin, dans l'appel III, `f(move(n)`, on force l'argument transmis à `f` à être une `rvalue`, d'où l'appel de `f(int &&)`.

En définitive, les trois appels de `f` provoquent les mêmes effets :

```
En I
-- Appel f(int &&)
---- Appel g(int &}
---- Appel h(int &)
---- Appel g(int &&}
En II
-- Appel f(int &)
---- Appel g(int &}
---- Appel h(int &)
---- Appel g(int &&}
En III
-- Appel f(int &&)
---- Appel g(int &}
---- Appel h(int &)
---- Appel g(int &&}
```

Les remarques déjà formulées à propos des risques que comporte l'utilisation d'un objet après qu'il a été transmis à une fonction comme référence à une `rvalue` s'appliquent également ici. Si `n` était un objet comportant des parties dynamiques, la fonction `f` pourrait tirer parti du fait qu'il est considéré comme temporaire. C'est d'ailleurs ce que feront le constructeur par déplacement et l'affectation par déplacement.

Exercice 169

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
void f(int && p)
{ cout << "-- Entree dans f -     p = " << p << endl ;
  p++ ;
  cout << "-- Sortie de f -       p = " << p << endl ;
}
int h (int n) { return n ;}
int main()
{ int n = 20 ;
  cout << "-- En A de main -      n = " << n << endl ;
  f(h(n)) ;       // appel I
  cout << "-- En B de main -      n = " << n << endl ;
  f(move(n)) ;    // appel II
  cout << "-- En C de main -      n = " << n << endl ;
  f(10) ;         // appel III
  cout << "-- En D de main, 10 vaut = " << 10 << endl ;
}
```

Solutions

Dans l'appel I, `h(n)` est une référence à la `rvalue` temporaire fournie par `h` (une copie de la valeur reçue en argument), qui se trouve transmise à `f`. Dans `f`, cette valeur temporaire est incrémentée de 1, ce que montrent les deux affichages. Mais, une fois revenu dans la fonction `main`, on constate naturellement que la variable `n` n'a pas changé de valeur.

En revanche, dans l'appel II, `f(move(n))`, on transmet à `f` une référence à `n`, considérée comme une `rvalue`. Dans `f`, `p` représente donc la référence à cette variable qui se trouve incrémentée de 1. Une fois revenu dans la fonction `main`, on constate bien que `n` vaut `21`.

Enfin, dans l'appel III, `f(10)`, c'est une copie temporaire de la constante `10` qui est transmise à `f`. Fort heureusement, la constante `10` utilisée dans la fonction `main` n'est pas affectée.

En définitive, nous obtenons les résultats suivants :

```
-- En A de main -      n = 20
-- Entrée dans f -     p = 20
-- Sortie de f -       p = 21
-- En B de main -      n = 20
-- Entrée dans f -     p = 20
-- Sortie de f -       p = 21
-- En C de main -      n = 21
-- Entree dans f -     p = 10
-- Sortie de f -       p = 11
-- En D de main, 10 vaut = 10
```

---

1. Cet exemple (artificiel) montre bien les conséquences qu'entraînent les faits de transformer explicitement par `move` une `lvalue` en une `rvalue` et d'en transmettre la référence à une fonction attendant précisément une référence à une `rvalue`.
2. Si nous avions défini `h` de cette manière :

   ```
   int & h (int n)
   ```

   l'appel `f(h(n))` conduirait à une erreur de compilation due à ce que `h(n)` est une `lvalue` et donc qu'elle ne peut plus correspondre à `f(int&&)`. La même remarque s'applique à la fonction `h`.

Exercice 170

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std;
class point ;
class point
{ public :
  point(int abs, int ord)
```

```
    { x=abs ; y=ord ;
      cout << "-- Construction  " << x << " " << y <<  "    "
<< this << endl ;
    }
    point (const point & p)
    { x=p.x ; y=p.y  ;
      cout << "-- Constr copie  " << x << " " << y << " de "
<< &p << " vers " << this << endl ;
    }
    point (point && p)
    { x=p.x ; y=p.y  ;
      cout << "-- Constr deplac " << x << " " << y << " de "
<< &p << " vers " << this <<  endl ;
    }
    ~point()
    { cout << "-- Destruction   " << x << " " << y << "    "
<< this << endl ;
    }
    int x, y ;   // public ici pour simplifier
} ;
point g ()          { cout <<   "---- Appel g\n" ; return
point(0,0) ; };
point h (point p) { cout <<  "---- Appel h\n" ; return p ; }
int main()
{  point p1 (6, 6) ;     cout << "-- EN I -----\n" ;
   point p2 (p1) ;       cout << "-- EN II ----\n" ;
   point p3 (g()) ;      cout << "-- EN III ---\n" ;
   point p4 (h(p3)) ;    cout << "-- EN IV ----\n" ;
   point p5 (move(p1)) ; cout << "-- EN V -----\n" ;
}
```

Notez bien que cet exercice est essentiellement destiné à montrer le mécanisme de choix entre construction par copie et construction par déplacement. En pratique, l'existence d'opérations de déplacement ne se justifierait que dans le cas de classes comportant des parties dynamiques.

La construction de `p1` fait naturellement appel au constructeur usuel. Celle de `p2` fait appel au constructeur par recopie.

Dans `p3(g())`, on pourrait s'attendre à ce qu'il y ait une création temporaire du `point(0,0)`, qui serait fournie comme `rvalue` au constructeur par déplacement afin de créer `p3`. En fait, le même résultat est obtenu de façon plus rapide en enlevant son caractère temporaire à `point(0,0)` et en faisant directement le point `p3`. Un résultat similaire pourrait être constaté si l'on

créait un point de cette façon :

```
point pp (point(0,0))
```

L'appel `p4(h(p3))` conduit à transmettre d'abord à `h` une copie temporaire de `p3`, correspondant à l'argument `p` de `h`. Comme la valeur de retour de `h(h(p3))` est temporaire, elle est considérée comme `rvalue` et donc transmise au constructeur par déplacement de `point`, pour créer `p4`. La copie temporaire est alors détruite.

Dans l'appel `p5(move(p1))`, on force la construction par déplacement de `p5`, à partir de `p1`.

En définitive, nous obtenons les résultats suivants :

```
-- Construction  6 6    0x6afee0                  // p1 (6, 6)
-- EN I -----
-- Constr copie  6 6 de 0x6afee0 vers 0x6afed8    // p2 (p1)
-- EN II ----
---- Appel g                                     // p3 (g())
-- Construction  0 0    0x6afed0
-- EN III ---
--  Constr  copie   0  0  de  0x6afed0  vers  0x6afee8      //  p4
(h(p3))
---- Appel h
-- Constr deplac 0 0 de 0x6afee8 vers 0x6afec8
-- Destruction   0 0    0x6afee8
-- EN IV ----
--  Constr  deplac  6  6  de  0x6afee0  vers  0x6afec0     //  p5
(move(p1))
-- EN V -----
-- Destruction   6 6    0x6afec0                // fin main
-- Destruction   0 0    0x6afec8
-- Destruction   0 0    0x6afed0
-- Destruction   6 6    0x6afed8
-- Destruction   6 6    0x6afee0
```

Exercice 171

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std;
class point
{ public :
  point(int abs, int ord)
  { x=abs ; y=ord ; cout << "-- constr        " << x << " "
<< y << "    " << this << endl ;
  }
  point (const point & p)
  { x=p.x ; y=p.y  ;
    cout << "-- constr copie  " << x << " " << y << " de "
<< &p << " vers " << this <<  endl ;
  }
  point (point && p)
  { x=p.x ; y=p.y ;
    cout << "-- constr deplac " << x << " " << y << " de  "
<< &p << " vers " << this << endl ;
  }
  point & operator = (const point & p)
  { x=p.x ; y=p.y  ;
    cout << "-- affect copie  " << x << " " << y << " de "
<< &p << " vers " << this << endl ;
    return * this ;
  }
  point & operator = (point && p)
  { x=p.x ; y=p.y  ;
    cout << "-- affect deplac " << x << " " << y << " de "
<< &p << " vers " << this << endl ;
    return *this ;
  }
  ~point()
  { cout << "-- Destruction   " << x << " " << y << "    "
<< this << endl ;
  }
  int x, y ;   // public ici pour simplifier
} ;
void f (point & p) { cout << "-- f(point &}    " << p.x << "
" << p.y << "    "  << &p <<  endl ;
                    }
void f (point && p){ cout << "- f(point &&}    " << p.x << "
" << p.y << "    " << &p << endl ;
                    }
point g(int n) { return point(n, n) ; }
point h(point p) { return p ; }
int main()
{ point p1 (1, 2) ; point p2 (2, 5) ;
  cout << "-- En I ------\n" ;
  f(p1) ;       cout << "-- En II -----\n" ;
```

```
    f(g(2)) ;     cout << "-- En III ----\n" ;
    p1 = g(4) ;   cout << "-- En IV -----\n" ;
    f(p1=p2) ;    cout << "-- En V ------\n" ;
    p1=p2=g(8) ;  cout << "-- En VI -----\n" ;
    f(move(p1)) ; cout << "-- En VII ----\n" ;
  }
```

Solutions

La construction des points `p1` et `p2` est classique. Il en va de même de l'appel `f(p1)` qui transmet simplement à `f(point &)` la référence de `p1`.

L'appel `f(g(2)` conduit d'abord à la création par `g` du `point(2,2)` temporaire, représenté par `g(2)`, `rvalue` dont la référence est fournie directement à `f(point&&)`. Après retour de `f`, ce point est détruit.

Dans `p1=g(4)`, il y a également création par `g` du `point(4,4)` temporaire, représenté par `g(4)`, `rvalue` dont la référence est fournie à l'affectation par déplacement. Après affectation, ce point est détruit.

L'appel `f(p1=p2)` provoque une affectation classique `p1=p2`, dont le résultat est transmis par référence à `f(point &)`.

Avec `p1=p2=g(8)`, il y a d'abord création du `point(8,8)` temporaire, représenté par `g(8)`, `rvalue` dont la référence est fournie à l'affectation par déplacement. Le résultat de cette dernière est, quant à lui, transmis par référence à l'affectation classique.

Enfin, dans `f(move(p1))`, on force l'appel de `f(point&&)`, en convertissant la `lvalue`, `p1`, en une `rvalue`.

En définitive, on obtient ces résultats :

```
-- constr        1 2    0x6afed0                    // p1
-- constr        2 5    0x6afec8                    // p2
-- En I ------
-- f(point &}    1 2    0x6afed0                    // f(p1)
-- En II -----
-- constr        2 2    0x6afed8                    // f(g(2)
- f(point &&}    2 2    0x6afed8
-- Destruction   2 2    0x6afed8
-- En III ----
-- constr         4 4     0x6afee0                      // p1 =
g(4)
```

```
-- affect deplac 4 4 de 0x6afee0 vers 0x6afed0
-- Destruction   4 4    0x6afee0
-- En IV -----
-- affect copie  2 5 de 0x6afec8 vers 0x6afed0      // f(p1=
p2)
-- f(point &}    2 5    0x6afed0
-- En V ------
-- constr        8 8    0x6afee8                   // p1 = p2
= g(8)
-- affect deplac 8 8 de 0x6afee8 vers 0x6afec8
-- affect copie  8 8 de 0x6afec8 vers 0x6afed0
-- Destruction   8 8    0x6afee8
-- En VI -----
- f(point &&}     8 8     0x6afed0                        //
f(move(p1))
-- En VII ----
-- Destruction   8 8    0x6afec8
-- Destruction   8 8    0x6afed0
```

Exercice 172

Énoncé

On dispose de la structure suivante :

```
struct coord
{ int x, y, z ;
}
```

Utiliser cette structure pour créer une classe `point3d` dans laquelle les coordonnées (entières) sont conservées dans une structure de type `coord` allouée dynamiquement. On dotera cette classe :

- d'un constructeur classique et d'une méthode `affiche` ;
- des constructeurs par recopie et par déplacement ;
- de l'affectation par recopie et par déplacement.

Écrire un petit programme de test de cette classe.

Solutions

Ici, nous conserverons l'adresse de la structure dynamique dans un pointeur nommé `adp`, de type `unique_ptr<coord>`, qui sera initialisé lors de la construction à l'aide de la fonction `make_unique`.

Le constructeur par recopie et l'affectation par recopie sont classiques.

Les opérations de déplacement (construction ou affectation) doivent toutes les deux fabriquer un nouvel objet identique à celui reçu en argument, sachant que ce dernier ne sera plus utilisé par la suite. Elles peuvent éviter de créer une nouvelle partie dynamique de type `coord` pour le nouvel objet, en se contentant de recopier le pointeur du premier. Ici, il nous suffira donc de transférer la propriétété du pointeur `adp` de `p` vers l'objet courant, en forçant l'affectation à l'aide de la fonction `move` :

```
adp = move(p.adp) ;
```

Naturellement, comme pour l'affectation par copie, on se contentera de ne rien faire lorsque l'on se trouve en présence de l'affectation d'un objet à lui-même.

On pourrait penser qu'il est également nécessaire de donner la valeur `nullptr` au pointeur `adp`, afin que la libération de l'objet devenu temporaire ne risque pas d'entraîner la libération de la partie dynamique dont on transfère la propriété au nouveau point. En fait, cette action est déjà réalisée par l'opérateur d'affectation par déplacement de la classe `unique_ptr<point>`. En revanche, une telle « mise à zéro » serait indispensable si l'on utilisait un pointeur usuel (`point *`).

Voici ce que pourrait être notre classe `point3d`. Ici, nous avons ajouté quelques instructions d'affichage dans certaines fonctions.

```
#include <iostream>
#include <memory>
using namespace std;
struct coord
{ int x, y, z ;
} ;
```

```
class point3d
{ public :
   point3d (int x=0, int y=0, int z=0)
   { adp = make_unique<coord> () ;
     (*adp).x = x ; (*adp).y = y ;  (*adp).z = z ;
   } ;
  void affiche()
  { cout << "-- Coordonnées : " << adp->x << " " << adp->y <<
" "
                                  << adp->z << endl ; }
  point3d (const point3d & p)              // constructeur de
recopie
  {  cout << "-- Construction copie \n" ;
     adp = make_unique<coord> () ;
    (*adp).x = (p.adp)->x ; (*adp).y = (p.adp)->y ; (*adp).z =
(p.adp)->z ;
  }
  point3d (point3d && p)                   // constructeur de
déplacement
  {  cout << "-- Construction déplacement \n" ;
     adp = move(p.adp) ;
     // p.adp = nullptr ;    n'est pas utile ici
  }
   point3d & operator = (const point3d & p) // affectation
copie
  { cout << "-- Affectation copie\n" ;
    if (&p != this)
     { (*adp).x = (p.adp)->x ; (*adp).y = (p.adp)->y ;
       (*adp).z = (p.adp)->z ;
     }
   return *this ;
  }
   point3d & operator = (point3d && p)      // affectation
déplacement
  { cout << "-- Affectation deplacement\n" ;
     if (&p != this)
     {  adp = move (p.adp) ;
                                          // p.adp = nullptr ;
n'est pas utile ici
     }
     return *this ;
  }
  private :
    unique_ptr<coord> adp = nullptr ;
} ;
point3d  origine ()
 { cout << "--appel origine\n" ; return point3d (0, 0, 0) ; }
int main()
```

```
{ point3d p1(3, 5) ; p1.affiche() ;         // construction
usuelle
  point3d p2(p1) ;   p2.affiche() ;       // construction par
recopie
   point3d p3(5, 5) ; p3.affiche() ;         // construction
usuelle
  p1 = p3 ;          p1.affiche() ;      // affectation copie
   p3 = origine() ;    p3.affiche() ;         // affectation
déplacement
   p3 = move(p2) ;     p3.affiche() ;       // p2 n'est plus
utilisable
    point3d p4 (origine()) ; p4.affiche(); // construction
déplacement
}
```

Voici les résultats fournis par ce programme :

```
-- Coordonnées : 3 5 0
-- Construction copie
-- Coordonnées : 3 5 0
-- Coordonnées : 5 5 0
-- Affectation copie
-- Coordonnées : 5 5 0
--appel origine
-- Affectation déplacement
-- Coordonnées : 0 0 0
-- Affectation déplacement
-- Coordonnées : 3 5 0
--appel origine
-- Coordonnées : 0 0 0
```

Exercice 173

Énoncé

Écrire un patron de fonction nommé `echange` permettant d'échanger les valeurs de deux objets de type `T`, la classe `T` étant supposée dotée d'un constructeur par déplacement et d'une affectation par déplacement.

Les arguments de la fonction `echange` devront naturellement être transmis par référence, comme dans une version « classique » de cette fonction d'échange, lorsqu'on ne cherche pas à faire appel à la sémantique de déplacement :

```
template <class T> void echange ( T & a, T & b)
{ T tempo (a) ;    // construction de tempo, par copie de a
  a = b ;          // affectation de b à a
  b = tempo ;      // affectation de tempo à b
}
```

Notez déjà que la première instruction est une construction par copie et non une affectation.

Si l'on veut profiter de l'existence d'opérations de déplacement dans la classe `T` pour optimiser le code, on remplacera simplement les deux opérations par copie (construction et affectation) par les opérations correspondantes par déplacement :

```
template <class T> void echange ( T & a, T & b)
{ T tempo  (move (a)) ;
  a = move (b) ;
  b = move (tempo) ;
}
```

Exercice 174

Énoncé

a. Quels résultats fournit le programme suivant :

```
#include <iostream>
#include <memory>
using namespace std ;

template <class T> void f(T&& n) { cout << "-- f(T&&)\n" ; }
void g (int && n)          { cout << "-- g(int &&)\n" ;
}       // I
void g (int & n)            { cout << "-- g(int &)\n" ;
}        // II
void g (const int & n)  { cout << "-- g(const int &)\n" ;
}  // III
int gh() { return 2 ; }
```

```
int main()
{ int n=10 ;
  const int p = 5 ;
  g(n) ; g(p) ; g(3) ; g(gh()) ;
  f(n) ; f(p) ; f(3) ; f(gh()) ;
}
```

b. Que se passerait-il si l'on supprimait la troisième définition (III) de `g` ?

c. Que se passerait-il si l'on supprimait, en outre, la deuxième définition (II) de `g` ?

Solutions

a.

Les quatre appels de la fonction `g` sont classiques (ils sont commentés dans les résultats). Rappelons que les appels `g(3)` et `g(gh())` conduisent tous les deux à l'appel de `g(int&& n)`, puisque les arguments sont des `rvalue`. Cependant, le premier se fait en transmettant à `f` l'adresse d'une copie de la constante littérale `5`, tandis que le second se fait en transmettant directement l'adresse du résultat temporaire fourni par `gh`.

En revanche, dans le patron de fonction `f`, la notation `T&&` correspond à une référence universelle, ce qui lui permet d'être appliquée indifféremment à une `lvalue` ou à une `rvalue`. Tous les appels de `f` sont donc acceptés par cette unique fonction :

```
-- g(int &)        // appliqué à une value modifiable
-- g(const int &)  // appliqué à une lvalue non modifiable
-- g(int &&)        // appliqué à une rvalue, copie du litéral
3
--  g(int  &&)           // appliqué directement à la rvalue
temporaire gh()
-- f(T&&)          // appliqué à une lvalue modifiable
-- f(T&&)          // appliqué à une lvalue non modifiable
-- f(T&&)          // appliqué à la copie du littéral 3
--  f(T&&)              // appliqué directement à la rvalue
temporaire gh()
```

b.

En revanche, si l'on supprimait la troisième défintion de `g`, l'appel `g(p)` conduirait à une erreur de compilation, due à ce qu'aucune des deux

fonctions g restant définies ne peut être appliquée à une lvalue constante.

c.

Si l'on supprimait en plus la deuxième définition de g, l'appel g(n) conduirait à son tour à une erreur de compilation, due à ce que la fonction restante ne peut pas s'appliquer à une lvalue.

C'est cette dernière situation, résumée ci-dessous, qui illustre le mieux la notion de référence universelle : dans un patron de fonction, la notation T&& peut correspondre à n'importe quoi, alors que int&& ne peut correspondre qu'à une rvalue.

```
template <class T> void f(T&& n) { ..... }
void g (int && n)                { ..... }
int gh() { return 2 ; }
int main()
{ int n=10 ;
  const int p = 5 ;
   // g(n) ; g(p) ;                    // appels rejetés en
compilation
  g(3) ; g(gh()) ;                 // appels g(int &&)
  f(n) ; f(p) ; f(3) ; f(gh()) ;  // appels f(T&&)
}
```

Exercice 175

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;

template <class T> void f(T&& n) { cout << "-- f(T &&)\n" ;
}
template <class T> void f(T& n)  { cout << "-- f(T &)\n" ; }
template <class T> void f(const T & n)
{ cout << "-- f(const T &)\n" ; }
void g (int && n)       { cout << "-- g(int &&)       " ; }
void g (int & n)        { cout << "-- g(int &)        " ; }
void g (const int & n)  { cout << "-- g(const int &)  " ; }
```

```
int gh() { return 2 ; }

int main()
{ int n=10 ;
  const int p = 5 ;
  g(n) ; f(n) ;
  g(p) ; f(p) ;
  g(3) ; f(3) ;
  g(gh()) ; f(gh()) ;
}
```

Il s'agit en fait d'une adaptation du programme du précédent exercice, dans lequel nous avons ajouté deux surdéfinitions du patron de fonction `f` correspondant à des arguments de type `T&` et `const T&`.

Dans ce cas, un appel tel que `f(n)` conduit au choix du patron `f(T&)` et non `f(T&&)`, car le premier est plus spécialisé. Les mêmes considérations s'appliquent à l'appel `f(p)`, où `f(const T&)` est plus spécialisé que `f(T&)`. En définitive, les résultats montrent ici une parfaite symétrie entre les fonctions `f` et les fonctions `g`.

```
-- g(int &)        -- f(T &)
-- g(const int &)  -- f(const T &)
-- g(int &&)       -- f(T &&)
-- g(int &&)       -- f(T &&)
```

Exercice 176

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
#include <memory>
#include <iostream>
using namespace std ;

void f ( int & n )           { cout << "f par lvalue
```

```
modifiable\n"; }
void f ( const int & n )   { cout << "f par lvalue non
modifiable\n"; }
void f ( int && n )       { cout << "f par rvalue\n"; }
template< class T > void essai ( T && n )
{ cout << "--- sans forward, ni move : " ;
  f (n) ;
  cout << "--- avec forward :          " ;
  f( forward <T> (n) );
  cout << "--- avec move :             " ;
  f( move (n) ) ;
}

int main()
{ cout << "-- Appel essai avec rvalue :\n";
  essai (5);
  cout << "-- Appel essai avec lvalue modifiable :\n";
  int p = 5;
  essai (p) ;
  cout << "-- Appel essai avec lvalue non modifiable :\n";
  const int q = 25 ;
  essai (q) ;
 }
```

Solutions

Dans le patron de fonction :

```
essai ( T && n )
```

`n` est une référence universelle qui peut correspondre indifféremment à une `lvalue` ou une `rvalue`. Mais on sait que, dans le corps de `essai`, `n` redevient toujours une `lvalue`. La fonction `forward`, appliquée à `n`, permet de conserver la nature (`rvalue` ou `lvalue`) de `n` et, en conséquence, d'appeler la fonction correspondante.

En définitive, nous obtenons les résultats de la page suivante.

```
-- Appel essai avec rvalue :
--- sans forward, ni move : f par lvalue modifiable
--- avec forward :          f par rvalue
--- avec move :             f par rvalue
-- Appel essai avec lvalue modifiable :
--- sans forward, ni move : f par lvalue modifiable
--- avec forward :          f par lvalue modifiable
--- avec move :             f par rvalue
```

```
-- Appel essai avec lvalue non modifiable :
--- sans forward, ni move : f par lvalue non modifiable
--- avec forward :          f par lvalue non modifiable
--- avec move :             f par lvalue non modifiable
```

Exercice 177

Énoncé

On dispose d'une classe `A` possédant :

- un constructeur par déplacement :

  `A (A &&)`

- un opérateur d'affectation par déplacement :

  `A & operator = (A &&)`

On souhaite créer une classe `B` dérivée de `A` comportant une partie dynamique destinée à contenir une chaîne de caractères ; pour cela, on lui prévoira deux nouveaux membres privés :

```
unique_ptr<char[]> adn ;    // pointeur sur un tableau de
char
int lg ;                   // longueur de la chaîne
```

Écrire le constructeur par déplacement et l'opérateur d'affectation par déplacement de cette classe `B`.

Solutions

Le constructeur par déplacement de `B` peut s'appuyer sur le constructeur par déplacement de `A` et ne s'occuper que des nouveaux membres introduits dans la classe dérivée :

```
    B (B && p) : A(p)
    { lg = p.lg ;
      adn = move (p.adn) ;
       // p.adn = nullptr ;   n'est pas utile ici car adn est
unique_ptr
    }
```

En revanche, on sait que l'opérateur d'affectation d'une classe dérivée, lorsqu'il est défini, ne fait pas appel automatiquement à l'opérateur d'affectation de la classe de base. Il est nécessaire d'y faire appel explicitement :

```
B & operator = (B && p)
{ point * ad1 = this ;
  point * ad2 = &p ;
  *ad1 = move(*ad2) ;    // force l'affection depl sur partie
A de B
  lg = p.lg ;
  adn = move (p.adn) ;
   // p.adn = nullptr ; n'est pas utile ici, car adn est
unique_ptr
}
```

Remarque

Comme indiqué en commentaire dans les deux opérations de déplacement, il n'est pas nécessaire ici d'affecter la valeur `nullptr` à `p.adp`, car cela est déjà réalisé par l'affectation par déplacement de la classe `unique_ptr<char[]>` dans les affectations `adn=move(p.adn)`. En revanche, si `adn` était un pointeur usuel, cette action serait indispensable pour éviter de prendre le risque que l'emplacement pointé soit détruit au moment de la destruction de l'objet temporaire `p`.

Exercice 178

Énoncé

L'exercice 151 a fait réaliser un patron de classes permettant de manipuler des vecteurs dynamiques dont les éléments sont d'un type quelconque. Compléter ce patron de manière à ce que ces classes disposent dorénavant d'un constructeur par déplacement, ainsi que d'une affectation par déplacement.

Solutions

Le constructeur par déplacement aura pour en-tête :

```
template <class T> vect<T>::vect (vect<T> && v)
```

Il doit créer un nouveau vecteur identique à `v`, en profitant de ce que `v` est temporaire, en réutilisant au maximum le vecteur associé et en se contentant de recopier la valeur du pointeur `adr` correspondant. Naturellement, le champ `nelem` doit être recopié. Il est également nécessaire d'affecter la valeur `nullptr` à `adr`, pour éviter que l'emplacement pointé ne soit détruit lors de la destruction du temporaire `v`. (Comme nous l'avons vu dans certains des exercices précédents, cette opération ne serait pas nécessaire si l'on avait affaire à un pointeur de type `unique_ptr` ou `shared_ptr`.) En définitive, voici ce que pourrait être ce constructeur :

```
template <class T> vect<T>::vect (vect<T> && v)
{ adr = move (v.adr) ;
  nelem = v.nelem ;
   v.adr = nullptr ;            // indispensable ici
}
```

L'affectation par déplacement est écrite de façon semblable :

```
 template <class T> vect<T> & vect<T>::operator = (vect<T> &&
v)
 { if (this != &v)                // on ne fait rien pour a=a
     { nelem = v.nelem ;
       adr = move (v.adr) ;
       v.adr = nullptr ;          // indispensable ici
     }
   return * this ;
}
```

À titre indicatif, voici un exemple de petit programme utilisant le nouveau patron de classes ainsi obtenu, ainsi qu'une nouvelle fonction indépendante nommée `crpointconst` :

```
vect<int> crpointconst (int n, int val)
{ vect<int> p(n) ;
  for (int i=0 ; i<n ; i++) p[i] = val ;
  return p ;
}
main()
{  int i ;
   vect <int> v1(5) ; vect <int> v2(10) ;
```

```
   for (i=0 ; i<5 ; i++) v1[i] = i ;
   cout << "v1 = " << v1 << "\n" ;
   for (i=0 ; i<10 ; i++) v2[i] = i*i ;
   cout << "v2 = " << v2 << "\n" ;
   vect <int> v3 (move(v1)) ;   // Ne plus utiliser v1 ensuite
   cout << "v3 = " << v3 << "\n" ; // v2 = v1 ;
                                      // provoquerait erreur
exécution
   v2 = move(v3) ;              // v3 n'existe  plus
   cout << "v2 = " << v2 << "\n" ;
   vect <int> v4 = crpointconst(3, 5) ;
   cout << "v4 = " << v4 << "\n" ;
   v2 = crpointconst (2, 0) ;
   cout << "v2 = " << v4 << "\n" ;
}

v1 = <0 1 2 3 4 >
v2 = <0 1 4 9 16 25 36 49 64 81 >
v3 = <0 1 2 3 4 >
v2 = <0 1 2 3 4 >
```

Merci d'avoir choisi ce livre Eyrolles. Nous espérons que sa lecture vous a été utile et vous aidera pour mener à bien vos projets.

Nous serions ravis de rester en contact avec vous et de pouvoir vous proposer d'autres idées de livres à découvrir, des nouveautés, des conseils ou des événements avec nos auteurs.

Intéressé(e) ? Inscrivez-vous à notre lettre d'information.

Pour cela, rendez-vous à l'adresse go.eyrolles.com/newsletter ou flashez ce QR code (votre adresse électronique sera à l'usage unique des éditions Eyrolles pour vous envoyer les informations demandées) :

Vous êtes présent(e) sur les réseaux sociaux ? Rejoignez-nous pour suivre d'encore plus près nos actualités :

Eyrolles Web Dev et Web Design

Merci pour votre confiance.

L'équipe Eyrolles

P.S. : chaque mois, 5 lecteurs sont tirés au sort parmi les nouveaux inscrits à notre lettre d'information et gagnent chacun 3 livres à choisir dans le catalogue des éditions Eyrolles. Pour participer au tirage du mois en cours, il vous suffit de vous inscrire dès maintenant sur go.eyrolles.com/newsletter (règlement du jeu disponible sur le site).

Pour suivre toutes les nouveautés numériques du Groupe Eyrolles, retrouvez-nous sur Twitter et Facebook

@ebookEyrolles

EbooksEyrolles

Et retrouvez toutes les nouveautés papier sur

@Eyrolles

Eyrolles